

H.C. Pennington
TRS-80® DISK
& OTHER MYSTERIES



THE "HOW TO" BOOK OF DATA RECOVERY

TRS-80 DISK AND OTHER MYSTERIES

TRS-80 INFORMATION SERIES VOLUME 1

BY H.C. PENNINGTON
Illustrations by the Author



Copyright (c) 1979 by Harvard C. Pennington

FIRST EDITION

FIRST PRINTING - NOVEMBER 1979
SECOND PRINTING - JANUARY 1980

All rights reserved. Reproduction or use, without express permission, of editorial or pictorial content, in any manner, is prohibited. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher and the author assume no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

PUBLISHED BY:

INTERNATIONAL JEWELRY GUILD, INC.

IJG COMPUTER SERVICES

569 N. Mountain Ave.

Upland, California 91768 U.S.A.

ISBN # 0-936200-00-6



Radio Shack and TRS-80 are the registered trademarks of the TANDY Corporation

CONTENTS

PREFACE	iii
INTRODUCTION	1
HEXADECIMAL - BINARY - DECIMAL	3
READING & USING SUPERZAP 2.0	9
"SUPERZAP" FUNCTIONS	9
"SUPERZAP" COMMANDS	13
SPECIAL COMMANDS	13
SPECIAL SYMBOLS	14
"SUPERZAP" DISPLAY FORMAT	14
EXAMPLES	15
SUPERZAP 3.0	20
OTHER UTILITIES	24
RSM-2D	24
MONITOR 3	24
DEBUG	25
DIRCHECK	25
LMOFFSET	28
OPERATING SYSTEMS	29
TRSDOS 2.1	29
TRSDOS 2.2	29
VTOS 3.0	29
NEWDOS 2.1	30
FUTURE DEVELOPMENTS	30
DISK ORGANIZATION	32
THE DIRECTORY	36
THE 'GAT' SECTOR	36
THE 'HIT' SECTOR	38
THE 'FPDE/FXDE' SECTORS	43
DECODING THE DIRECTORY ENTRIES	45
DECODING THE EXTENTS	49
PASSWORDS & OTHER TRIVIA	52
DATA RECOVERY PROCEDURES & TECHNIQUES	54
THE 'SHELL GAME' ON DISK	54
USING "SUPERZAP" ON SINGLE DRIVE SYSTEM	55
BUILDING A BUFFER TRACK	55
FILES - STRUCTURES & TYPES	56
ASCII 'BASIC' PROGRAM FILES	56
BINARY 'BASIC' PROGRAM FILES	57
EDITOR/ASSEMBLER SOURCE FILES	59
OBJECT CODE FILES	60
SYSTEM FILES	62
ELECTRIC PENCIL FILES	63
MACRO-80 FILES	64
DATA RECOVERY	66
RECOVERING 'HASH' CODES	66
RECOVERING A 'KILLED' FILE	66
RECOVERING 'DISK WON'T READ/BOOT'	68
RECOVERING 'CLOBBERED' DIRECTORY	69
RECOVERING 'UNREADABLE' DIRECTORY	70

RECOVERING 'ELECTRICALLY' DAMAGED DISK	71
RECOVERING A PHYSICALLY DAMAGED DISK	72
RECOVERING 'BAD PARITY' ERROR	72
RECOVERING A DIRECT STATEMENT IN FILE	74
ASCII 'BASIC' PROGRAM FILE	75
BINARY 'BASIC' PROGRAM FILE	75
RECOVERING DATA FILES	77
ASCII FILES	77
RANDOM FILES	78
RECOVERING 'ELECTRIC PENCIL' ERRORS	81
'DOS ERROR 22'	81
'LOST' FILE ON DISK	82
'LOST' FILE IN MEMORY	83
RECOVERING 'FILE AREA FULL' ERROR	84
ELECTRIC PENCIL GOODIES	84
CORRECTING THE GAT & HIT SECTORS	86
SOME THINGS YOU CAN DO	88
MAKING ELECTRIC PENCIL FILES IN 'BASIC'	88
LOADING 'BASIC'/ASCII FILES INTO PENCIL	88
MAKING 'PENCIL' FILES INTO 'BASIC' FILES	89
CONVERTING 'DATA TYPES' IN RANDOM FILES	89
CONVERTING DATA IN ASCII FILES	90
MAKING 'BASIC' PROGRAMS 'UNLISTABLE'	90
ADDING COMMANDS TO "SUPERZAP"	90
APPENDIX A	
GLOSSARY	1
LEVEL II 'BASIC' TOKENS	10
TRSDOS 2.2 DIRECTORY HEX DUMP	11
NEW DOS 2.1 DIRECTORY HEX DUMP	16
VTOS 3.0 DIRECTORY HEX DUMP	21
APPENDIX B	
DISK DRIVE MAINTENANCE	1
SUGGESTED READING	3
MURPHY'S LAW & OTHER COROLLARIES	4
ORDERING NEW DOS & SUPERZAP	5
"SEARCH" PROGRAM DOCUMENTATION	6

MAP INDEX

DISK MAP (TRS DOS 2.2)	35
DIRECTORY MAP (TRACK 11 - ALL SYSTEMS)	37
GAT SECTOR MAP (TRACK 11, SECTOR 0).....	40
GRANULE ALLOCATION MAP	41
HIT SECTOR MAP (TRACK 11, SECTOR 1)	42
FPDE/FXDE SECTOR MAP (TRACK 11, SECTOR 0-9)...	44
DIRECTORY ENTRY MAP	45

PREFACE

I'll never forget how I first met Harvard C. Pennington, the author of "TRS-80 DISK AND OTHER MYSTERIES". I was attending a meeting of our local TRS-80 users' group when I happened to glance over at one of our Radio Shack managers. He appeared to be short of breath. On further examination I saw that, in fact, he was being garroted by a disk cable assembly. The garroter, I found out later, was Harv.

Harv has since taken less drastic measures in attempting to find the answers to some of the perplexing problems that appear in TRSDOS and other Radio Shack and non-Radio Shack Utility and applications software. He has gone from violence to investigative writing. The results of his research are presented here in "TRS-80 DISK AND OTHER MYSTERIES".

Is this a worthwhile book? To use one of Harv's expressions, "Hell yes!" (You'll find other salty expressions herein, but they just liven up the book.) But seriously, TRS-80 users, this is not only a worthwhile book but a great book. It's great for two reasons: It presents information on TRS-80 disk organization and file management that can be found nowhere else! Secondly, it is available when you need it - now!

The book discusses how disks are organized, how space is allocated, how files are located on disk, and the tools that one may use to look at disk files and directories. Not only does it provide a general discussion of these topics, however, but it also gives clear information how to FIX disk problems such as lost files, Electric Pencil bugs and other snafus.

This is a clearly presented book packed with good disk information. My advice to you is to get it, use it, and do not approach Mr. Pennington while carrying a disk cable assembly.

William Barden Jr.

INTRODUCTION

I have been programming for a very short time and having applied myself to the task, it seems I have acquired some knowledge that others would like me to share with them.

No doubt you have been told that you cannot do certain things with the TRS-80 --- like 'BOOT' a 'BASIC PROGRAM' because you need 'BASIC' to load a program --- or that you cannot lock out the break key without messing up the I-O routines --- or that you cannot defeat the 'LIST' and 'LLIST' commands. You have been told wrong. All of these things can be done! I have been able to do all of the above with little or no trouble. The ONLY limitation you have is your own imagination.

Of course, there is no fool-proof way of protecting anything because some determined soul will puzzle out the most obscure and hidden method and reveal it to the world ... just as I'm doing here.

The following is a result of endless hours of gazing at the CRT, countless disk dumps, and many hours of cross checking. Now that you have developed a certain amount of respect for my efforts, as a result of reading the above, we will proceed.

...Oh, yes. This couldn't have been done without an incredible program called "SUPERZAP". It is a product of APPARAT Inc., of Denver Colorado. You may purchase a copy of this program with the NEW DOS operating system from your local software dealer. Ordering information is in the appendix at the end of this tome. You will find that "SUPERZAP" is indispensable if you are going to take the voyage to the bottom of the disk.

The following people have contributed, in one way or another, to my somewhat limited store of knowledge or to the completion of this book. I would like to have them stand and take a bow:

Bill Barden	Jim Farvour	Ron Markle
Jim Lauletta	C.I.	Michael Shrayer
Dick Schubert	Stu Nims	Dennis Fagan
Bob Thorpe		

To the above: Thank you from the bottom of my CPU.

SOME KIND WORDS ON THE TRS-80

On the whole the TRS-80 is a pretty neat machine. In fact, I love my TRS-80. Just a few short years ago, a computer with the power and capability of the TRS-80 would have cost several hundreds of thousands of dollars, required an air conditioned room of considerable proportions and a staff to operate it. Certainly, the TANDY Corporation deserves all the credit possible for the development, production and distribution of this magic machine. TANDY Corporation, I SALUTE YOU!

SOME NOT SO KIND WORDS

Like all large corporations, the TANDY Corporation, seems to have continued success in spite of itself. The initial success of the TRS-80 was, I suspect, beyond the wildest dreams of anybody at Tandy. Since there was no way to measure their success against a similar product, at a similar price and with similar distribution, who is to say how successful they really were. It is my contention that they were only about fifty to sixty percent as successful as they could have been!

Very quickly, as a result, an attitude of "don't-tell-us, we'll-tell-you" developed. The general quality of follow-on support, development and software was abysmal. Information about the workings of the system was (and is) a carefully and jealously guarded secret. It's as if "WE", the users, "couldn't possibly know a damn thing or figure it out" and only the High Priests of Fort Worth, when they deem it propitious, will tell us what we need to know. So, if I tend to excoriate (a fancy word meaning, "give 'em hell") the TANDY Corporation (Radio Shack), it is only because I would like to see them turn around their damn superior attitude and realize that the thousands of you out there are doing more than you are being given credit for and should be listened to. Instead of 'THEM' telling you; 'YOU' tell them.

A case in point is the APPARAT Corporation. Here are two guys in Denver, without the resources or the staff, working at other endeavours, and they have single-handedly revised, corrected and enhanced the operating system (TRSDOS 2.1) over one year ago! They got it into release with almost no bugs! When they did find bugs, they admitted it and sent out corrections immediately. They provided the user with the tools to 'get into the disk' ("SUPERZAP") and make the fixes. You will shortly receive word on a new Radio Shack break through - TRSDOS 2.3! Tell me, does this mean there are bugs in 2.2 or does this mean that there were some things in 2.2 that were (to use the words of Radio Shack) '...not fully implemented'?

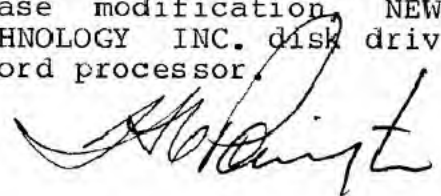
It is my experience that when you find it difficult (if not impossible) to admit that you have made a mistake, you should try to cover it up with what George Orwell might term "CRAPSPEAK".

ABOUT THIS BOOK

Just by reading this book, one might get the impression that the only thing the TRS-80 is good for is to fix errors that are created on the machine! Not true! Not a single day goes by that I do not use my TRS-80 for some useful and productive purpose. Occasional errors are just a small part of the day to day experience. It is only when you cannot fix those errors that they begin to dominate the 'computer experience'.

It is my wish that you will, as a result of this book, be able to make your TRS-80 one hundred percent productive and enjoyable.

This book was written, composed, directed, choreographed, and produced on a TRS-80 with 48K RAM, upper/lower case modification, NEWDOS operating system, four MICROCOMPUTER TECHNOLOGY INC. disk drives, Spinterm Printer and the 'ELECTRIC PENCIL' word processor.



To Kip and Trista

Knowledge is a commodity that can be exchanged for time.

1.0 IS IT A NUMBER OR A LETTER

Most of the numbers we will be using, in our journey through the disk, are HEXADECIMAL numbers. The following is a brief outline of the HEXADECIMAL numbering system. If you are totally unfamiliar with 'HEX' numbers, I would suggest you get a copy of William Barden's "How to Program Microcomputers". Chapter two will make you an expert. In the meantime the following will acquaint you with the HEXADECIMAL numbering system.

The computer does all of its thinking in BINARY numbers. Since we human beings don't 'naturally' think in BINARY numbers the computer does a number on the numbers and presents the information we need in DECIMAL numbers. However, DECIMAL numbers are too long when we need to fit large numbers onto the video display - especially if we need to put a lot of them on the screen at one time. Also the computer can convert BINARY to HEXADECIMAL very easily and quickly.

HEXADECIMAL is usually shortened to 'HEX' and sometimes to 'H'. There are other methods used to indicate that the numbers being used are 'HEX' and we'll get to that later.

1.1 BINARY

You have ten fingers and those that study such matters tell us that for this reason we just 'naturally' think in tens. To represent each finger we have a symbol. The symbols we use are:

(figure 1.1)

THE TEN DECIMAL SYMBOLS

0 1 2 3 4 5 6 7 8 9

The computer, on the other hand, has only two 'fingers' ('ON' and 'OFF') and therefore naturally thinks in twos and only needs two symbols to represent the numbers. The symbols used by the computer are:

(figure 1.2)

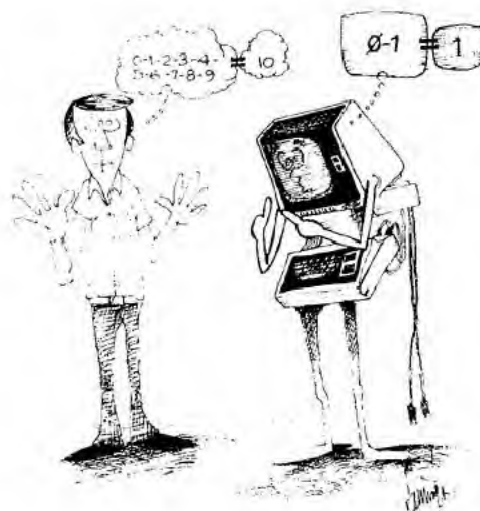
THE TWO BINARY SYMBOLS

0 1

In order to represent ALL of the numbers, we use a system that puts a VALUE on WHERE the number symbols are in relation to each other.

(figure 1.3)

...you have 10
fingers



... the
computer only
has two 'fingers' ...

When we humans get to the symbol '9' we have to start using our number symbols over again. We move the '1' one place to the left and start over with the '0' symbol on the right. When we move the '1' to the left, we also assign a different value to it DEPENDING ON HOW MANY PLACES TO THE LEFT WE MOVE IT.

(figure 1.4)

10 = ()	4 = ()
9 = ()	3 = ()
8 = ()	2 = ()
7 = ()	1 = ()
6 = ()	0 = ()
5 = ()			

From the above figure, you can visualize the relationship between decimal numbers and the values they represent. A peculiarity of humans is that we never think of 'ZERO' as a number. Actually 'ZERO' is the FIRST number and should always be thought of in that manner; i.e., start counting with 'ZERO' as your first number. If you count the fingers on your hand, starting with 'ZERO', you will only get to nine before you run out of fingers to count. THIS IS THE WAY THE COMPUTER COUNTS. It ALWAYS starts with zero!

Now we are ready to investigate the computer's method of counting. The computer only has two fingers -- 'ON' and 'OFF' -- simply

described, this is due to the fact that a digital device, such as a computer, can only detect one of two conditions, 'ON' or 'OFF'. If you will remember from the above discussion, a human starts using the symbols over after '9' and has 10 symbols to work with. The computer starts using the symbols over again after '1'. In other words, when the computer gets to '1' he runs out of number symbols and starts over by moving the '1' one place to the left and starts over with the 'ZERO' symbol on the right

(figure 1.5)

BINARY NUMBER	TALLEY	DECIMAL NUMBER
1010	= ()	= 10
1001	= ()	= 9
1000	= ()	= 8
111	= ()	= 7
110	= ()	= 6
101	= ()	= 5
100	= ()	= 4
11	= ()	= 3
10	= ()	= 2
1	= ()	= 1
0	= ()	= 0

Let's take a closer look at that BINARY number '1010'. First of all, it has four 'places'. (Count the digits, there are four of them.) Each 'place' represents a "times two" multiplication. We'll convert this BINARY number '1010', to DECIMAL by multiplying each 'place' by it's 'place value' and adding the results.

(figure 1.6)

BINARY	DECIMAL
1 0 1 0	
	'place' number 1 is units 0 X 1 = 0 'place' number 2 is twos 1 X 2 = 2 'place' number 3 is fours 0 X 4 = 0 'place' number 4 is eights 1 X 8 = 8 +
	10 (DECIMAL)

If we add the results of our multiplications (0 + 2 + 0 + 8 = 10) we will have converted our BINARY number (BASE 2) to a DECIMAL number (BASE 10).

1.2 HEXADECIMAL

Now we'll tackle HEXADECIMAL (BASE 16). The computer needs a method

of representing large numbers in a small space. BINARY is easy to convert to HEXADECIMAL (for the computer, at least). The HEXADECIMAL system uses 16 symbols to represent the 16 numbers and then, just like BINARY and DECIMAL numbers, we move the '1' one place to the left and start using the number symbols over again.

The following are the HEXADECIMAL numbers with their DECIMAL equivalents:

(figure 1.7)

HEX		DECIMAL	HEX		DECIMAL
0	=	0	8	=	8
1	=	1	9	=	9
2	=	2	A	=	10
3	=	3	B	=	11
4	=	4	C	=	12
5	=	5	D	=	13
6	=	6	E	=	14
7	=	7	F	=	15

It was decided (by whom, I don't know) to use letters for the additional HEXADECIMAL symbols, since letters and numbers are already on the keyboard. As a result, we get numbers that look like this: '1A' or 'FF'. You will find that using HEXADECIMAL numbers is not as inconvenient as you might suspect. After a couple of days they become very familiar indeed.

Here is an instant replay of the above figures in HEXADECIMAL, BINARY and DECIMAL. This time I have shortened it up a bit because the numbers from '0' to '9' are the same in HEXADECIMAL as in DECIMAL.

(figure 1.8)

HEX	TALLY	DECIMAL	BINARY
20	= ()	= 32	= 100000
1A	= ()	= 26	= 11010
15	= ()	= 21	= 10101
10	= ()	= 16	= 10000
F	= ()	= 15	= 1111
A	= ()	= 10	= 1010
5	= ()	= 5	= 101

As you can see, we run out of number symbols after we get to 'F' and just as in every other numbering system, we start over by moving the '1', one place to the left and placing the zero in the 'units' position.

So when you see 10 (HEX) or 10 (BINARY) or 10 (DECIMAL) you know I am

talking about 3 different VALUES of 'one-zero'.

There is one more thing I would like to tell you about that will come in handy as you progress through this book. From time to time you will need to convert a binary 'bit record' into it's HEX value. This is easier than you might think. Consider the following:

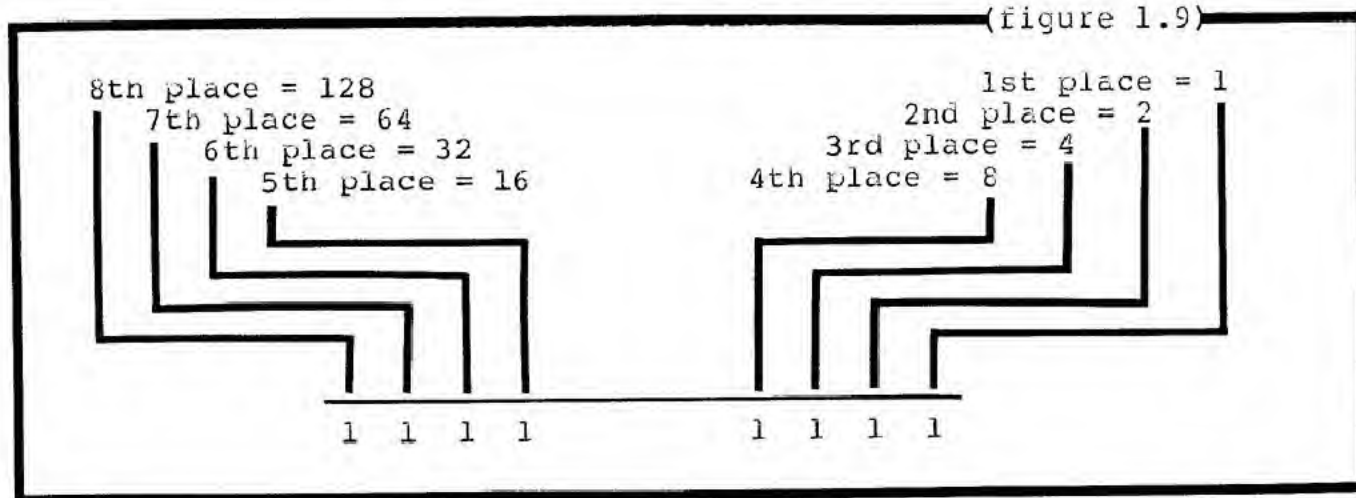
FF (HEX) = 11111111 (BINARY)

It looks complicated dosen't it? Well, it's not. It is the essence of simplicity. The way to solve any problem is to break it down into manageable chunks. This problem is no different. First we'll take the BINARY number, which, in this case is 8 bits or one byte, and break it up into what is termed, in the trade, as a 'NYBBLE'. A 'NYBBLE' is 4 bits. Let's break the above example into 'NYBBLE's:

FF (HEX) = 1111 1111

Now if you will get your lightning-quick-bear-trap-mind into remember mode, you will recall the 'place values' for the BINARY numbers. If you don't recall, then I'll review it for you.

(figure 1.9)



If we take each group of four bits (NYBBLE), we can easily figure out the HEX value, since we can easily remember the HEX numbers from '0' to 'F'. Adding up the BINARY numbers on the right we arrive at the following:

$$\begin{array}{r} 8 \times 1 = 8 \\ 4 \times 1 = 4 \\ 2 \times 1 = 2 \\ 1 \times 1 = 1 \end{array} +$$

15 (DECIMAL) = F (HEX)

Do it again for the left side then combine the two HEX values and you've got it!

Let's do it again with a different BINARY number, just to make sure. Suppose we need the HEX value for the Binary Value '00101101'. Here are our NYBBLES: 0010 and 1101.

LEFT SIDE			
8	X	0	= 0
4	X	0	= 0
2	X	1	= 2
1	X	0	= 0 +

2 (DECIMAL)

2 (DECIMAL) = 2 (HEX)

LEFT SIDE + RIGHT SIDE = 2D (HEX)

RIGHT SIDE			
8	X	1	= 8
4	X	1	= 4
2	X	0	= 0
1	X	1	= 1 +

13 (DECIMAL)

13 (DECIMAL) = D (HEX)

There, isn't that simple? With very little practice you should be able to convert BINARY to HEX and vice-versa with very little trouble.

THE BAD DREAM



2.0 READING & USING "SUPERZAP"

"SUPERZAP" is unique in several ways. First it has its own disk I/O routines and does not require that a DOS be in drive zero to perform miracles. Second, it will read ANYTHING that is readable, regardless of its 'PROTECT' status. Third, it will recover almost every imaginable type of error condition.

In addition, it has a 'BACKUP' routine that will make umpty-ump tries when it encounters an incorrect or electrically damaged sector before it gives up. Then, it allows you to try again as many times as you want!

Wait! There's more... it will copy disk sectors, relocate disk sectors, allow modification of any byte or combination of bytes on the disk or in memory, move data from one disk sector to another and 'ZERO' disk sectors.

Now that you have a preview of what it can do, lets review the functions and commands of "SUPERZAP" so you can start using it today.

2.1 SUPERZAP FUNCTIONS

(figure 1)

APPARAT SUPERZAP 2.0

INPUT ONE OF THE FOLLOWING INSTRUCTIONS

'DD' OR NULL - DISPLAY DISK SECTOR
'PD' - PRINT DISK SECTORS
'DM' - DISPLAY MAIN MEMORY
'PM' - PRINT MAIN MEMORY
'VERIFY DISK SECTORS'
'ZERO DISK SECTORS'
'COPY DISK SECTORS'
'DISK BACKUP'
'COPY DISK DATA'

?

"SUPERZAP" menu display

We'll take each menu function in order. I would recommend that you get in front of the computer, 'RUN' "SUPERZAP" and try out each function and command as it is explained. This way you will become familiar with the operation of "SUPERZAP" that much quicker.

```
***** NOTE *****
**
** ENTER ALL MENU FUNCTIONS WITHOUT QUOTES OR APOSTROPHES **
**
*****
```

'DD' or 'NULL' - DISPLAY DISK SECTOR. You will use this function more than any other. You will be constantly looking at the sectors to verify or change something. 'NULL' in this case means press <ENTER>. Since this function is used so much Cliff (the author of "SUPERZAP") decided it would be nice to eliminate the constant typing of 'DD'. (Thank you, Cliff.) After you enter <ENTER> or 'DD' the computer will respond with the prompts in figure 2.2.

(figure 2.2)

```
RELATIVE DISK # (0 - 3)?  
TRACK # (HEX) (0 - 22)?  
SECTOR # (0 - 9)?
```

Answer 'RELATIVE DISK #' with the drive number you wish to work with. TRACK # you will notice, only allows you to answer with a number between zero and twenty-two. (Zero to twenty-seven if you are using a 40 track version.) YOU WILL HAVE TO CONVERT ALL DECIMAL NUMBERS TO HEXADECIMAL NUMBERS! SECTOR # is easy. There are 10 sectors numbered zero to nine. Pick a drive, track, and sector and go look at it. When you are tired of looking, press 'X' and the menu will magically reappear. There are more things we can do while in this function, but we'll come back to that later

'PD' - PRINT DISK SECTORS. This function is almost (I say, almost) exactly the same as the 'DD' function except that the sector(s) will be printed on your line printer and you may not modify anything. This function will ask you for one additional parameter: SECTOR COUNT. Enter the number of sectors you want printed out, hit <ENTER> and stand back.

```
***** NOTE *****  
**  
** ENTER 'SECTOR COUNT' IN DECIMAL! **  
**  
*****
```

This function beats the Radio Shack 'DISKDUMP/BAS' program all to smash. It will dump a 'PROTECTED' file without any of that 'FILE ACCESS DENIED' business. If you suddenly decide you want to halt the printing function, HOLD DOWN THE 'H' KEY UNTIL THE PRINTER STOPS.

'DM' - DISPLAY MAIN MEMORY. This does for RAM exactly what 'DD' does for the disk. The prompt will ask for the RAM address (in HEX) instead of the 'DISK', 'TRACK' and 'SECTOR'. Later on, when we discuss the many command features of 'DD', they will apply to this function also. Pressing 'X' will return you to the menu.


```

***** WARNING *****
**
** MODIFICATIONS (USING 'MODnn'), MADE TO MAIN MEMORY,**
** ARE COMPLETE AS SOON AS THE MODIFICATION APPEARS **
** ON THE SCREEN. Unlike modifying the disk, you do **
** not have the opportunity to cancel the change. **
**
*****

```

'PM' - PRINT MAIN MEMORY. This function duplicates the 'PD' function, only it works on RAM or ROM. Holding down the 'H' key will terminate the function just as with 'PD'.

'VERIFY DISK SECTORS'. This function will locate sectors that are write protected, sectors with parity errors, and sectors with physical damage.

You may select a 'PAUSE' option to halt the verification process each time a 'READ PROTECTED' sector is encountered. This will allow you to note those sectors for special attention later on.

It is especially useful in discovering where a specifically bad sector or sectors are on a 'flaky' disk when you need to recover 'lost' data. This function requires a sector count in decimal.

'ZERO DISK SECTORS'. From time to time, you will need to zero an entire sector or group of sectors. This would be a very tedious task indeed if you had to do it a byte at a time which, by the way is possible but certainly not desirable. If the sector you are zeroing is 'READ PROTECTED', you will be asked if you want that sector to remain 'READ PROTECTED'. A reply of 'Y' or 'N' (YES or NO) will determine the 'READ PROTECT' status of the zeroed sector. This function requires a sector count in decimal.

'COPY DISK SECTORS'. With this beauty, you can copy a single sector or group of sectors from one location to another or from disk to disk. When we have to reconstruct a file that the DOS has strewn all over the disk, you'll kiss the very envelope "SUPERZAP" came in. What would ordinarily have been a bitch to recover will be so easy, you will want to amaze your friends and neighbors with your wizardry.

Normally this function copies the sectors in ASCENDING track and sector order. However, if the lowest destination sector is within the range of the source sectors, the function will copy in DESCENDING order. This will occur automatically and the routine will compute the highest track and sector of each range BEFORE starting the copy.

This permits you to copy a group of sectors TO a location that starts WITHIN the group of sectors you are copying FROM.

The 'READ PROTECT' status of the destination sectors remains unchanged by the 'COPY SECTORS' function. This function requires a sector count in DECIMAL.

'DISK BACKUP'. Amazing! Simply amazing. This function simply backs up the disk. BUT WHAT A BACKUP! The routine is a sector-by-sector

backup and is S-L-O-W. But it is sure. It retrieves the sectors that are not readable by regular 'COPY' or 'BACKUP' routines. It will make a dozen or so tries to read a 'bad' sector and will give you an error message similar to figure 2.3 if it cannot accomplish its 'READ'.

(figure 2.3)

```
SECTOR READ ERROR
DRIVE 1 , TRACK 05 , SECTOR 0
SYSTEM ERROR CODE 04
PARITY ERROR
REPLY 'R' FOR RETRY, 'S' FOR SKIP ERROR SECTOR,
OR 'X' TO CANCEL FUNCTION?
```

Now you have several choices; (1) press 'X' and forget the whole thing, (2) press 'S' and 'SKIP' the bad sector (and come back to it later - but don't forget to make notes so you'll remember which sector or sectors were bad) or (3) press 'R' and re-enter the 'BACKUP' routine and try again. Many times the 'R' command will be effective and the BACKUP routine will successfully read the bad sector on the second or third try.

A 'READ' after every 'WRITE' is performed to verify that an accurate data transfer has taken place.

```
***** CAUTION *****
You must 'BACKUP' to a diskette that has been PREVIOUSLY
FORMATTED. The 'SOURCE' diskette and the 'DESTINATION'
diskette MAY NOT BE THE SAME - in other words, this func-
tion requires TWO DRIVES! The 'DESTINATION' diskette is
not tested for name or contents - if it is possible to
'WRITE' to that diskette, ALL DATA ON THE 'DESTINATION'
DISKETTE WILL BE WRITTEN OVER WITH THE DATA FROM THE
'SOURCE' diskette!
*****
```

'COPY DISK DATA'. This is similar to 'COPY DISK SECTORS' except that the function copies BYTES. Up to as many as 65,536 bytes at one time and as few as one byte. Here is another super function for recovering 'lost' data.

The same rules apply to the ASCENDING and DESCENDING track/sector/byte order of the 'COPY DISK DATA' function as the 'COPY DISK SECTORS' function.

The 'READ PROTECT' status of the destination sectors/bytes remains unchanged. A BYTE COUNT IS REQUIRED IN HEX!

2.2 SUPERZAP COMMANDS

When "SUPERZAP" is in the 'DISPLAY DISK SECTORS' or 'DISPLAY MAIN MEMORY' function, the program is constantly monitoring the input keys looking for one of the following commands:

- 'X' - Terminate the primary function.
- 'R' - Repeat display of the same sector or memory block.
- 'J' - Restart the same primary function.
- 'K' - ('DD' only) Same as 'J' except reinitializes the track and sector to be displayed on the same disk drive.
- 'H' - ('PD' and 'PM' only) Terminates PRINT function.
- '+' or ';' - Scroll forward one sector or memory block.
- '=' or '-' - Scroll backward one sector or memory block.

2.3 SPECIAL COMMANDS

The following commands only work in 'DISPLAY DISK SECTORS' and 'DISPLAY MAIN MEMORY'. They are used chiefly for MODIFYING the memory or disk a byte at a time. The commands are:

- 'MODnn' - Modify the byte in the currently displayed sector where 'nn' is a hexadecimal number representing the relative byte to be modified.
(See EXAMPLE 1 below for use of this command.)
- <ENTER> - AFTER modifying a byte or group of bytes <ENTER> causes the modification to be written to the disk.
- <SPACE BAR> - Current digit is not changed and modification position is advanced 1 digit.
- 'RIGHT ARROW' - Same as <SPACE BAR>
- 'LEFT ARROW' - Current digit is not changed and modification position is retarded 1 digit.
- <SHIFT>'RIGHT ARROW' - Modification position is advanced 4 digits.
- <SHIFT>'LEFT ARROW' - Modification position is retarded 4 digits.
- 'UP ARROW' - Modification position is retarded 1 line.
- 'DOWN ARROW' - Modification position is advanced 1 line.
- 'SCOPY' - ('DD' only) Move the displayed sector to a disk location to be specified.

2.4 SPECIAL SYMBOLS

During the modification of a byte of memory or disk some special symbols appear to mark the location of the line and byte you are working on.

These symbols are 'M', '+', '-', '*', and '/'. The 'M' will mark the line and will appear BETWEEN the first column of six digits on the left of the screen, and the first column of 4 hex digits representing the sector's contents (see figure 2.1 line '11460').

The '+', '-', '*', and '/' will appear NEXT to the group of four digits IN WHICH THE MODIFICATION WILL TAKE PLACE. The '+' symbol indicates that the first digit is the digit which will be modified. The '-' indicates the second modification digit, the '*' the third and the '/' the fourth. With these symbol indicators, you will be able to tell which digit you will modify next.

2.5 SUPERZAP DISPLAY FORMAT

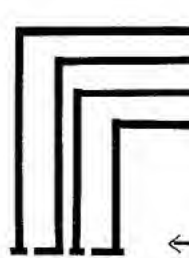
Before we can move on and actually demonstrate with some examples, you must first understand the display format of "SUPERZAP". Figure 2.4 is a typical sector display. The six digits on the far left side of figure 2.4, contains the following parameters (from left to right):

Position 1 ----- The disk drive used.
Positions 2 & 3 - The track being read.
Position 4 ----- The relative sector within the track.
Positions 5 & 6 - The relative byte count within the sector.

At the bottom of the sector, in the first group of digits, there is an extra digit in the seventh position. This '6' is APPARAT'S way of telling you that you are reading a 'READ' protected sector with the "SUPERZAP" program.

To the right of these six digits is a block of 32 digits in groups of 4. Each PAIR of digits represents a SINGLE BYTE. To the right of this is the ASCII representation of each byte. The 'dots' signify a space or 'unprintable' character. What's an 'unprintable' character? Just that. It's a valid ASCII character but there is no symbol that represents that character. The TRS-80 uses some of these characters for graphics symbols or space compression codes but they are not ASCII standard characters. Besides, if those positions were filled up with a bunch of 'garbage' characters, it would make the display that much harder to read.

A 'dot' can also represent a 'space'. There is a fine distinction between a 'space' and an 'unprintable' character. In BASIC program files, the 'unprintable' characters are 'next line pointers', 'EOR' markers, line numbers, and BASIC tokens. (More on 'tokens' later.). SOMETIMES there will be a HEX value in one of these and it will cause a character to be printed in the display. It will appear as if 'garbage' has crept into your program but don't despair; all is well. A 'space', on the other hand, is represented by the HEX character '20'. This is one case where nothing is something so look for it and don't be confused by it.



	Hexadecimal display of sector contents.								ASCII display of sector contents.
111400	5F00	0000	0053	5953	3020	2020	2053	5953SYS0.....SYS
111410	EB29	210E	0F00	0022	FFFF	FFFF	FFFF	FFFF	.)!....."
111420	0000	0000	0000	0000	0000	0000	0000	0000
111430	0000	0000	0000	0000	0000	0000	0000	0000
111440	M0000+	0000	0000	0000	0000	0000	0000	0000
111450	0000	0000	0000	0000	0000	0000	0000	0000
111460	0000	0000	0000	0000	0000	0000	0000	0000
111470	0000	0000	0000	0000	0000	0000	0000	0000
111480	0000	0000	0000	0000	0000	0000	0000	0000
111490	0000	0000	0000	0000	0000	0000	0000	0000
1114A0	0000	0000	0000	0000	0000	0000	0000	0000
1114B0	0000	0000	0000	0000	0000	0000	0000	0000
1114C0	0000	0000	0000	0000	0000	0000	0000	0000
1114D0	0000	0000	0000	0000	0000	0000	0000	0000
1114E0	0000	0000	0000	0000	0000	0000	0000	0000
1114F06	0000	0000	0000	0000	0000	0000	0000	0000

Typical "SUPERZAP" display of a SECTOR as it will appear on your video display. This particular sector is an example of a DIRECTORY SECTOR (Track 11, Sector 4).

2.6.1 EXAMPLE 1: 'MODnn'

To modify a byte or bytes in a particular sector, first select 'DD' or '<ENTER>' from the "SUPERZAP" menu. Answer the parameter questions with the drive, track and sector specifications of your choice. When the sector is displayed on the video monitor, TYPE: MOD42

```
***** NOTE *****
**
**  NOTHING WILL APPEAR ON THE DISPLAY OR
**  GIVE ANY INDICATION THAT ANYTHING IS
**  HAPPENING UNTIL YOU HAVE ENTERED THE
**  ENTIRE COMMAND.
**
*****
```

Magically, an 'M' will appear on the fifth line from the top and between the first six columns on the left and the first group of four digits on the right. In front of the second group of four digits on the right the '+' sign will also appear. (See figure 2.4)

We are now ready to MODIFY the display. You may enter any valid hexadecimal digit by simply typing the digit. Each time you press a valid key, the digit will be changed and the symbol in front of the group of four digits you are working on will be changed. These symbols give you an indication of where you are during the modification process.

If you wish to skip over a digit, press the <SPACE BAR> or the 'RIGHT ARROW'. The symbol indicator will change and each time you input four modifications or spaces, the '+' symbol will reposition in front of the next group of four digits.

When you have completed your modifications, hit <ENTER> and the modifications will be written to the disk. When the 'WRITE' is complete, you will get the prompt in figure 2.5.

(figure 2.5)

```
MODIFICATIONS COMPLETE.  
REPLY <ENTER> TO CONTINUE?
```

Upon pressing <ENTER>, the sector will again be 'READ' from the disk and displayed on the screen for your inspection. You may now visually verify that the changes have been made. You may modify any sector any number of times.

Now, by pressing the '+' (<SHIFT> is not necessary) you will scroll forward to the next sector and pressing the '-' key will scroll backward one sector.

Pressing 'R' will cause the primary function to be repeated. In this case it is 'DISPLAY DISK SECTORS'. Pressing 'R' causes the last sector specified to be read and displayed.

Pressing 'K' will allow you to specify another TRACK and SECTOR on the same drive and remain in the 'DD' function without having to go back to the menu.

Pressing 'J' is the same as 'K' except you may also re-specify the drive number as well as the track and sector without going back to the menu.

Pressing 'X' or entering 'X', DURING ANY PARAMETER SPECIFICATION, will return you to the menu.

Pressing 'Q' will cancel the 'MOD' function WITHOUT CHANGING THE DISK CONTENTS.

Now that you have been through these functions and commands, make a backup disk of your DOS and try out some of the things we've been over.

```
***** WARNING *****  
**  
** ALWAYS PRACTICE OR ATTEMPT DATA RECOVERY **  
** ON A BACKUP VERSION OF THE TROUBLE DISK. **  
** FAILURE TO DO SO CAN COST YOU VALUABLE **  
** DATA. **  
**  
*****
```

2.6.2 EXAMPLE 2: 'SCOPY'

'SCOPY' permits you to duplicate an entire sector to another location on the same disk or to another location on a different drive WHILE IN THE 'DD' MODE!

Suppose, for a moment, that you have attempted to read a sector that has bad parity and you get the 'BUFFER MAY CONTAIN ALL OR SOME OF SECTOR'S DATA' error message. Upon investigation, you determine that some of the bytes in the sector that are displayed are correct and you would like to preserve them so that they can be used to 'reconstruct' the damaged sector. TYPE: SCOPY. NOTHING WILL APPEAR ON THE SCREEN UNTIL THE ENTIRE COMMAND HAS BEEN ENTERED. You will, after typing 'SCOPY', get a prompt similar to figure 2.6.

(figure 2.6)

```
DRIVE 1 , TRACK 12 , SECTOR 9
IS TO BE COPIED TO
RELATIVE DISK # (0 - 3)?
TRACK # (HEX) (0 - 22)?
SECTOR # (0 - 9)?
```

After answering disk number, track number and sector number, the destination location will be checked, by the program, to make sure that the place you want to copy to is OK.

If the destination is 'flaky', you'll get another error message as to the cause of the condition. If the destination checks out, the 'WRITE' will be completed. You will then be prompted to hit <ENTER> to view the transferred sector at the new location.

When you attempt to do a recovery of a file or portions of a disk, it is a good idea to set up some 'BUFFER SECTORS'. This 'buffer' is simply a temporary storage place to put things while you're out there mucking around on the disk. You will also need to keep track of where you have put various sectors so that during the reconstruction you will not get mixed up. Another good practice is to reconstruct the file to yet another 'buffer area'. When the reconstruction is complete THEN transfer the reconstructed file or sectors to the original location.

2.6.3 EXAMPLE 3: 'COPY DISK DATA'

This function allows you to move blocks of data in the same manner that you move sectors. We can copy a single byte or group of bytes from one location to another.

Suppose we need to move a 32 byte directory entry from one sector to a different sector and position it at a different relative byte.

The following examples, with BEFORE and AFTER 'pictures', will illustrate the prompts and results:

(figure 2.7)

```

011400 5F00 0000 0053 5953 3020 2020 2053 5953 .....SYS0....SYS
011410 EB29 210E 0F00 0022 FFFF FFFF FFFF FFFF .)!....".....
011420 0000 0000 0000 0000 0000 0000 0000 0000 .....
011430 0000 0000 0000 0000 0000 0000 0000 0000 .....
011440 0000 0000 0000 0000 0000 0000 0000 0000 .....
011450 0000 0000 0000 0000 0000 0000 0000 0000 .....
011460 0000 0000 0000 0000 0000 0000 0000 0000 .....
011470 0000 0000 0000 0000 0000 0000 0000 0000 .....
011480 1000 0027 0044 4F53 4E4F 5445 5350 434C ...'.DOSNOTESPCL
011490 9642 9642 0400 0020 FFFF FFFF FFFF FFFF .B.B.....
0114A0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114B0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114C0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114E0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114F06 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

SOURCE SECTOR (Track 11, sector 4)

This is the sector that contains the data, beginning at relative byte 80 (HEX) that we wish to copy. We want to copy the 32 bytes beginning at relative byte 80 (HEX) to another sector.

The 'E5's contained in the example are for purposes of illustration only. An actual directory sector would contain zeros. (Figures 2.8 and 2.11)

(figure 2.8)

```

001300 E5E5 E5E5 E5E5 E5E5 E5E5 E5D5 E5E5 E5E5 .....
001310 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001320 E5E5 E5E5 E5E5 E5D5 E5E5 E5E5 E5E5 E5E5 .....
001330 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001340 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001350 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001360 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001370 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001380 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001390 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013A0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013B0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013C0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013D0 E5E5 E5E5 E5E5 E5E5 E5E5 E4E5 E5E5 E5E5 .....
0013E0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013F06 E5E5 E5E5 E5E5 E4E5 E5E5 E5E5 E5E5 E5E5 .....

```

DESTINATION SECTOR (BEFORE) (Track 1, sector 3)

This is the sector we wish to copy the data TO, beginning at relative byte 'E0' and continuing for the next 32 bytes.

Figure 2.9 is the "SUPERZAP" menu and the function input.

(figure 2.9)

```
INPUT ONE OF THE FOLLOWING INSTRUCTIONS
'DD' OR NULL - DISPLAY DISK SECTOR
'PD' - PRINT MAIN MEMORY
'DM' - DISPLAY MAIN MEMORY
'PM' - PRINT MAIN MEMORY
'VERIFY DISK SECTORS'
'ZERO DISK SECTORS'
'COPY DISK SECTORS'
'DISK BACKUP'
'COPY DISK DATA'
? COPY DISK DATA <ENTER>
```

The next prompt will request that the SOURCE, DESTINATION and BYTE counts be input. They will appear as in figure 2.10.

(figure 2.10)

```
PROVIDE SOURCE BASE INFORMATION
RELATIVE DISK # (0 - 3)? 0
TRACK # (HEX) (0 - 22)? 11
SECTOR # (0 - 9)? 4
RELATIVE BYTE # IN SECTOR (HEX, 00-FF)? 80

PROVIDE DESTINATION BASE INFORMATION
RELATIVE DISK # (0 - 3)? 0
TRACK # (HEX) (0 - 22)? 1
SECTOR # (0 - 9)? 3
RELATIVE BYTE # IN SECTOR (HEX, 00-FF)? E0
BYTE COUNT (HEX)? 20
```

Once the above parameters have been entered to copy the bytes from one location to another, we'll get the results shown in figure 2.11. Don't forget... THE BYTE COUNT IS IN HEXADECIMAL!

```

001300 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001310 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001320 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001330 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001340 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5D5 E5E5 .....
001350 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001360 E5E5 E5E5 E5E5 E5E5 E5D5 E5E5 E5E5 E5E5 .....
001370 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001380 E5E5 E5E5 E5D5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
001390 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013A0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013B0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013C0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013D0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....
0013E0 1000 0027 0044 4F53 4E4F 5445 5350 434C ...'.DOSNOTESPCL
0013F0 9642 9642 0400 0020 FFFF FFFF FFFF FFFF .B.B.....

```

DESTINATION SECTOR (AFTER) (Track 1, sector 3)

As you can see, the data is now in another sector and starts at a different relative byte.

2.7 SUPERZAP 3.0

About the time you think you have the 'last word' or the best of something, someone comes along and improves it. Yes, indeed, "SUPERZAP" has been improved (or enhanced). The MENU of "SUPERZAP" 3.0 now has an added function: 'DFS' - DISPLAY FIELD'S SECTORS.

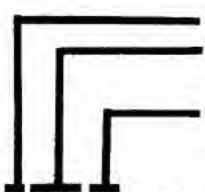
This will allow you to access a file WITHOUT KNOWING THE RELATIVE SECTOR NUMBER OF THE FILE OR ITS EXTENTS! It functions like 'DD' except you just specify a 'FILENAME' (with a password if passwords are required on that particular file) and the RELATIVE SECTORS OF THE FILE are displayed rather than the RELATIVE SECTORS OF THE DISK.

To invoke the function, type: DFS <ENTER>

The prompt will ask for the FILESPEC. Be sure to include the password if the file has one. The next prompt will ask for the relative sector (HEX) within the file. Remember the FIRST relative sector is SECTOR '0'!

The display format for 'DFS' is slightly different and will appear as in figure 2.12. 'DFS' uses standard BASIC mode 'RANDOM' I/O.

(figure 2.12)



'F' Indicates that 'DFS' is the function being used.
Relative sector displayed.
Relative byte

F00000	FFF4	6832	0093	3A20	4D41	494E	2F44	4953	...2...MAIN/DIS
F00010	4B20	4D45	4D4F	5259	2044	554D	502F	4D4F	K.MEMORY.DULP/MO
F00020	4449	4659	2052	4F55	5449	4E45	2E20	2056	DIFY.ROUTINE...V
F00030	4552	5349	4F4E	2032	2E30	0000	6964	008D	ERSION.2.0.....
F00040	2031	3034	3030	0029	6996	0041	24D5	C93A	.10400.)...A\$...:
F00050	208F	2041	24D5	2222	20CA	2031	3430	3A20	...A\$."...150:..
F00060	203A	9520	4258	D5F6	2841	2429	3A20	9200	...BX..(A\$):...
F00070	4E69	C800	8F20	4258	20D4	D534	3820	D220	N.....BX...48...
F00080	4258	D6D5	3537	20CA	2042	58D5	4258	CE34	BX..57...BX.BX.4
F00090	383A	2092	0072	69FA	008F	2042	58D4	D536	8:.....BX..6
F000A0	3520	D220	4258	D6D5	3730	20CA	2042	58D5	5...BX..70...BX.
F000B0	4258	CE35	353A	2092	0080	692C	0142	58D5	BX.55:.....,BX.
F000C0	CE42	583A	2092	00AF	695E	0193	3A20	2A2A	.BX:.....:**
F000D0	2A2A	2A2A	2A2A	2A20	5641	5249	4142	4C45	*****.VARIABLE
F000E0	2041	4C4C	4F43	4154	494F	4E20	494E	4849	.ALLOCATION.INHI
F000F0	4249	5445	4400	D569	9001	4432	2528	3129	BITED.....D2%(1)

The other enhancements are in the MODIFICATION mode. The new command is:

'ZTnn' (ZERO BYTES from the current modification location to 'nn' where 'nn' is a HEX number not exceeding 'FF').

This command functions like 'MODnn' in that NOTHING WILL APPEAR ON THE DISPLAY UNTIL 'ZT' IS ENTERED. Upon entering the 'ZT' portion of the command, 'ZT' will appear in column 7 and as you enter your HEX value, the value will also appear in column 7. Figure 2.13 will illustrate 'ZT' command display.

(figure 2.13)

F00000Z	FFF4	6832	0093	3A20	4D41	494E	2F44	4953	...2...MAIN/DIS
F00010T	4B20	4D45	4D4F	5259	2044	554D	502F	4D4F	K.MEMORY.DUMP/MO
F000208	4449	4659	2052	4F55	5449	4E45	2E20	2056	DIFY.ROUTINE...V
F00030F	4552	5349	4F4E	2032	2E30	0000	6964	008D	ERSION.2.0.....
F00040	2031	3034	3030	0029	6996	0041	24D5	C93A	.10400.)...A\$...:
F00050	208F	2041	24D5	2222	20CA	2031	3530	3A20	...A\$."...150:..
F00060	203A	9520	4258	D5F6	2841	2429	3A20	9200	...BX..(A\$):...
F00070	M4E69	C800	8F20	4258	20D4	+D534	3820	D220	N.....BX...48...
F00080	4258	D6D5	3537	20CA	2042	58D5	4258	CE34	BX..57...BX.BX.4
F00090	383A	2092	0072	69FA	008F	2042	58D4	D536	8:.....BX..6
F000A0	3520	D220	4258	D6D5	3730	20CA	2042	58D5	5...BX..70...BX.
F000B0	4258	CE35	353A	2092	0080	692C	0142	58D5	BX.55:.....,BX.
F000C0	CE42	583A	2092	00AF	695E	0193	3A20	2A2A	.BX:.....:**
F000D0	2A2A	2A2A	2A2A	2A20	5641	5249	4142	4C45	*****.VARIABLE
F000E0	2041	4C4C	4F43	4154	494F	4E20	494E	4849	.ALLOCATION.INHI
F000F0	4249	5445	4400	D569	9001	4432	2528	3129	BITED.....D2%(1)

In the above figure, you will observe that the 'MOD' symbol is at relative byte '7A' and that 'ZT', in column 7 is set for 'ZT8F'. This will ZERO ALL BYTES (from the 'MODnn' symbol) TO RELATIVE BYTE '8F' as in figure 2.14.

(figure 2.14)

```

F00000Z FFF4 6832 0093 3A20 4D41 494E 2F44 4953 ...2...MAIN/DIS
F00010T 4B20 4D45 4D4F 5259 2044 554D 502F 4D4F K.MEMORY.DUMP/MO
F00020B 4449 4659 2052 4F55 5449 4E45 2E20 2056 DIFY.ROUTINE...V
F00030F 4552 5349 4F4E 2032 2E30 0000 6964 008D ERSION.2.0.....
F00040 2031 3034 3020 0029 6996 0041 24D5 C93A .10400.)...A$...
F00050 208F 2041 24D5 2222 20CA 2031 3530 3A20 ...A$."...150:..
F00060 203A 9520 4258 D5F6 2841 2429 3A20 9200 ...BX..(A$):...
F00070 M4E69 C800 8F20 4258 20D4 0000 0000 0000 N.....BX.....
F00080 0000 0000 0000 0000 0000 0000 0000 0000 .....
F00090 +383A 2092 0072 69FA 008F 2042 58D4 D536 8:.....BX..6
F000A0 3520 D220 4258 D6D5 3730 20CA 2042 58D5 5...BX..70...BX.
F000B0 4258 CE35 353A 2092 0080 692C 0142 58D5 BX.55:.....,BX.
F000C0 CE42 583A 2092 00AF 695E 0193 3A20 2A2A .BX:.....:..**
F000D0 2A2A 2A2A 2A2A 2A20 5641 5249 4042 4C45 *****.VARIABLE
F000E0 2041 4C4C 4F43 4154 494F 4E20 494E 4849 .ALLOCATION.INHI
F000F0 4249 5445 4400 D569 9001 4432 2528 3129 BITED.....D2%(1)

```

I don't know about you, but I think that's pretty slick. Now I don't have to type in all those zeros to clean up a directory!

Suppose you accidentally enter the wrong number and wish to cancel the 'ZT' command? Easy. Just hit any invalid key, like a 'P' or a '\$' --- any one will do, and the display will respond with:

(figure 2.15)

```

F00000Z FFF4 6832 0093 3A20 4D41 494E 2F44 4953 ...2...MAIN/DIS
F00010T 4B20 4D45 4D4F 5259 2044 554D 502F 4D4F K.MEMORY.DUMP/MO
F00020B 4449 4659 2052 4F55 5449 4E45 2E20 2056 DIFY.ROUTINE...V
F00030F 4552 5349 4F4E 2032 2E30 0000 6964 008D ERSION.2.0.....
F00040 2031 3034 3030 0029 6996 0041 24D5 C93A .10400.)...A$...
F00050 208F 2041 24D5 2222 20CA 2031 3530 3A20 ...A$."...150:..
F00060 203A 9520 4258 D5F6 2841 2429 3A20 9200 ...BX..(A$):...
F00070 M4E69 C800 8F20 4258 20D4+D534 3820 D220 N.....BX...48...
F00080 4258 D6D5 3537 20CA 2042 58D5 4258 CE34 BX..57...BX.BX.4
F00090 383A 2092 0072 69FA 008F 2042 58D4 D536 8:.....BX..6
F000A0 3520 D220 4258 D6D5 3730 20CA 2042 58D5 5...BX..70...BX.
F000B0C 4258 CE35 343A 2092 0080 692C 0142 58D5 BX.55:.....,BX.
F000C0H CE42 583A 2092 00AF 695E 0193 3A20 2A2A .BX:.....:..**
F000D0E 202A 2A2A 2A2A 2A20 5641 5249 4142 4C45 *****.VARIABLE
F000E0C 2041 4C4C 4F43 4154 494F 4E20 494E 4849 .ALLOCATION.INHI
F000F0K 4249 5445 4400 D569 9001 4432 2528 3129 BITED.....D2%(1)

```

Look carefully in column 7 of figure 2.15 and you will notice that 'CHECK' now appears in the last five lines of column 7. The program will not allow you to make any more entries or modifications until the 'CHECK' error status is cleared. Now, type: <SHIFT> *. The entire command will be cleared and you can now start over.

'CHECK' was the next thing I was going to tell you about but I jumped the gun a little. 'CHECK' also works on 'MODnn' as well and will tell you when you have tried to input an invalid character.

"SUPERZAP" 3.0 also permits you to read up to 80 tracks! So, when drives with a bunch of tracks become available, all you little Zappers, out there, will be able to "ZAP" anything with any track configuration. The program also has provisions for backing up large track configurations to smaller track configurations.



3.0 OTHER UTILITIES

Besides "SUPERZAP", there are other utilities that may come in handy or that you may use instead of "SUPERZAP". Of all the utilities, that I know of, there are none that compare with "SUPERZAP" for ease of operation or versatility in the recovery process. The other utilities I am referring to are:

- RSM-2D (Small Systems Software)
- MONITOR 3 (ACS)
- DEBUG (Radio Shack)
- DIRCHECK (Apparat)
- LHOFFSET (Apparat)

Since many will ask, "Can I use this really neat program I bought from the MICRO-SUPER-80-SOFT-TRON Company in Elephant Breath, Ohio, to do the same thing so I don't have to buy 'SUPERZAP'?" The answer is, "Beats me lieutenant, I'm not the regular crew-chief." I will review each program that I have any knowledge of and explain how the program MIGHT be used in the data recovery process IF it can be used.

3.1 "RSM-2D"

'RSM-2D' is a product of Small Systems Software. It is well written and bug free. The documentation is not excellent but by comparison it is a cut above most.

RSM-2D is one of a family of machine language monitor programs for the TRS-80 and is based on widely used S-100 monitor programs. The '2D' version allows you to 'read' and 'write' disk sectors directly. It incorporates a special printer routine that outputs to the TRS-232 printer interface, also sold by Small Systems Software, as well as the standard parallel printer port.

The two commands added to the disk version, are 'L' (LOAD) and 'S' (SAVE). 'L' will load specified sectors into a specified block of memory and 'S' will write a specified block of memory to specified sectors on the disk.

In using RSM-2D for data recovery, you will find it adequate but cumbersome. This is due to the fact that you must always be working between disk AND memory. In addition, you will not have the advantage of a formatted display that shows you the ASCII as well as the HEX, in a sector by sector presentation. You may view the sectors in ASCII or HEX, but not both at the same time. You must also remember where the sector boundaries are in memory, in order to perform 'read's and 'write's to disk.

The software is reliable and you will not experience difficulty in its use except for the inconvenience of having to do some extra bookkeeping on sector boundary locations and interpretation between the HEX and ASCII display formats.

3.2 "MONITOR 3"

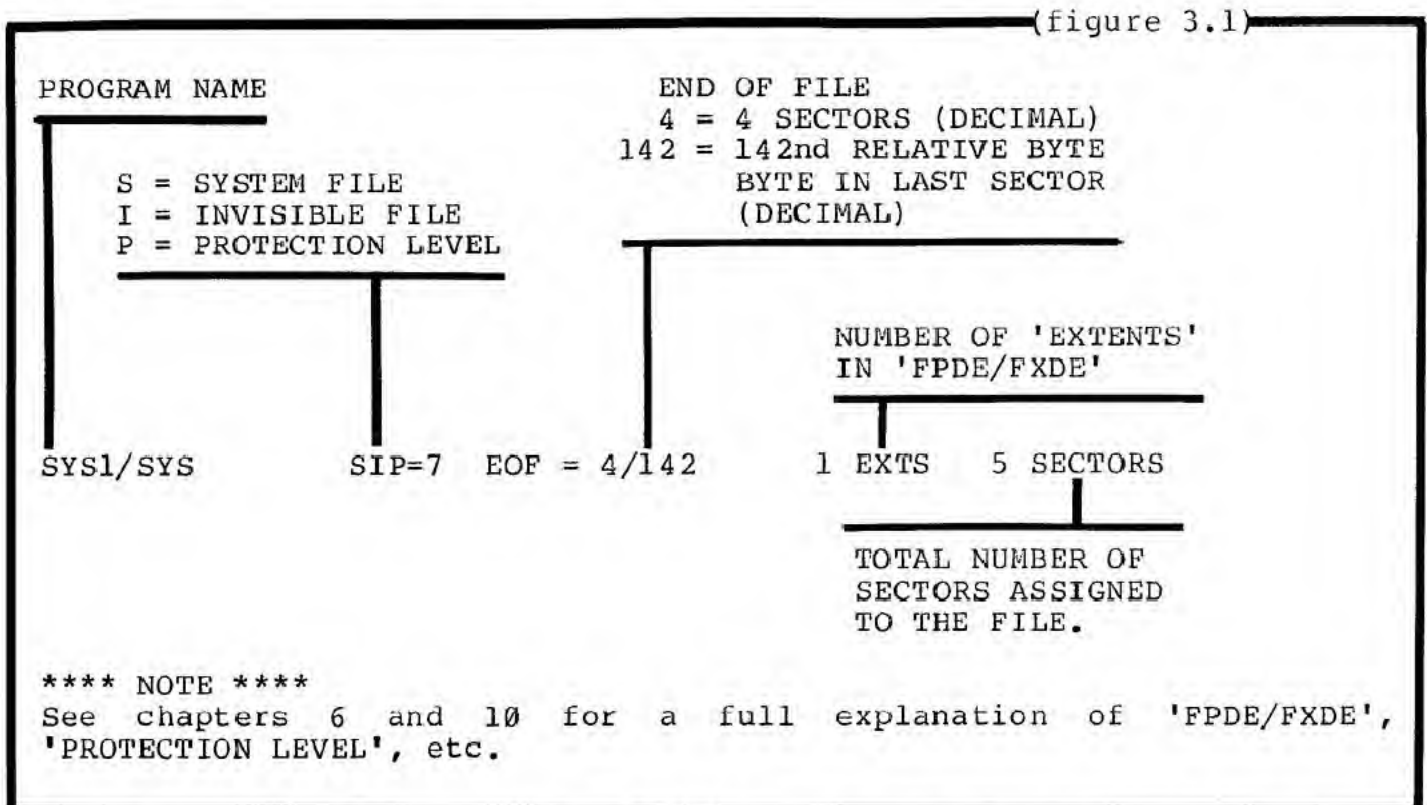
This is also a well written monitor program, but it does not have adequate disk I/O to be of any value in the data recovery process on the disk. I have seen a 'MONITOR 4' advertised by the same people that reads and writes to disk. I do not know if it has provisions that will allow you to repair the disk.

3.3 "DEBUG"

This is the standard Radio Shack monitor program that is included on every TRSDOS operating system disk. It is one of the Shack's better pieces of software and deserves mention, as such, but has no disk I/O capability and therefore has no application in the actual recovery process. It may be used however, after recovering a machine language load module to check and debug the module, after it is loaded into memory.

3.4 "DIRCHECK"

This is a utility program included in Apparat's NEW DOS+ package. It is an invaluable tool for checking the directory for errors. In addition it prints (to the video or line printer) an alphabetized listing of the directory entries, the 'END OF FILE' (EOF), in 'SECTOR/BYTE' format, the number of 'EXTENTS' for each file and the total number of sectors allocated (instead of 'GRANULES') to that file. Figure 3.2 is an example of the output of 'DIRCHECK' as it would look with errors in the directory sectors. Figure 3.1 is an explanation of that output.



In the next figure (3.2) there is a number beside each file name. This number is the 'DEC' (Directory Entry Code) for that file name. A complete explanation of the 'DEC' is contained in chapter 6. Also see figure 6.13 for details on decoding the 'DEC',

NEWDOS+ 07/15/79

```

64      BAD "HIT" SECTOR BYTE
BASIC/CMD 84 PRIMARY ENTRY HAS BAD CODE IN "HIT" SECTOR
00      ***** GRANULE FREE, BUT ASSIGNED TO FILE(S)
          00 BOOT/SYS
1E      ***** GRANULE LOCKED OUT, BUT FREE
1F      ***** GRANULE LOCKED OUT, BUT FREE
20      ***** GRANULE FREE, BUT ASSIGNED TO FILE(S)
          84 BASIC/CMD
36      ***** GRANULE ALLOCATED BUT NOT ASSIGNED TO ANY FILE
37      ***** GRANULE ALLOCATED BUT NOT ASSIGNED TO ANY FILE
05      ***** GRANULE ALLOCATED, BUT ASSIGNED TO MULTIPLE FILES
          83 SUPERZAP/PCL
          C7 DISKORG/PCL

```

BASIC/CMD	I	EOF = 6/231	2 EXTS	10 SECTORS
BOOT/SYS	SIP=6	EOF = 19/119	2 EXTS	5 SECTORS
COPY/CMD	IP=6	EOF = 4/253	1 EXTS	5 SECTORS
DIR/SYS	SIP=5	EOF = 16/0	1 EXTS	10 SECTORS
DIRCHECK/CMD		EOF = 12/136	3 EXTS	15 SECTORS
DISKORG/PCL		EOR 18/211	2 EXTS	20 SECTORS
FORMAT/CMD	IP=6	EOF = 14/8	1 EXTS	15 SECTORS
SYS0/SYS	SIP=7	EOF = 12/93	1 EXTS	15 SECTORS
SYS11/SYS	QIP=7	EOF = 4/142	1 EXTS	5 SECTORS
SYS12/SYS	SIP=7	EOF = 4/236	1 EXTS	5 SECTORS
SYS13/SYS	SIP=7	EOF = 3/9	1 EXTS	5 SECTORS
SYS2/SYS	SIP=7	EOF = 4/52	1 EXTS	5 SECTORS
SYS3/SYS	SIP=7	EOF = 4/76	1 EXTS	5 SECTORS
SYS4/SYS	SIP=7	EOF = 4/186	1 EXTS	5 SECTORS
SYS5/SYS	SIP=7	EOF = 4/203	1 EXTS	5 SECTORS
SYS6/SYS	SIP=7	EOF = 13/33	1 EXTS	5 SECTORS
SUPERZAP		EOF = 21/38	4 EXTS	25 SECTORS

43 FREE GRANULES 0 LOCKED-OUT GRANULES

NEWDOS DIRECTORY CHECK & LIST COMPLETED

It doesn't take a Radio Shack store manager to figure out that this recap of the directory's 'GAT' and 'HIT' errors is an extremely valuable tool in detecting existing errors in the directory.

'DIRCHECK' should be run on every disk in your library, from time to time, just to make sure some 'fatal' error isn't lurking and just waiting to clobber some really important data.

When an error exists in the 'BOOT/SYS' or if the directory track

has become NON-READ PROTECTED, or a 'PARITY' error exists in 'BOOT/SYS' or in any directory sector, 'DIRCHECK' will terminate with the following message:

FUNCTION TERMINATED DUE TO ERROR

You will still be able to read all of the sectors with "SUPERZAP". You must correct these defects before you will be able to run 'DIRCHECK'. The recovery procedures are described in chapter 10.

The following are the errors that are detected and printed by 'DIRCHECK' and what they mean.

3.4.1 BAD "HIT" SECTOR BYTE

A 'HASH' code exists in the 'HIT' sector when there should be none. The number, at the far left, represents the RELATIVE byte address of the bad code in the 'HIT' sector. Replace the offending code with '00'. The number beside the program name is the 'DEC' for that program. See figure 6.13 for details on decoding the 'DEC'.

3.4.2 PRIMARY ENTRY HAS BAD CODE IN "HIT" SECTOR

A 'HASH' code exists in the 'HIT' sector that is the WRONG code for the corresponding 'FPDE/FXDE' entry. The number, at the far left, represents the RELATIVE byte address of the incorrect code in the 'HIT' sector. Replace the 'HASH' code with the correct 'HASH' code.

3.4.3 GRANULE FREE BUT ASSIGNED TO FILE(S)

A 'GRANULE' has been allocated and there is no file using that granule. The number at the far left is the relative 'GRANULE' number in the GRANULE ALLOCATION TABLE. Replace the offending code with the proper code for that GRANULE. The number beside the program name is the 'DEC' for that program. See figure 6.13 for details on decoding the 'DEC'.

3.4.4 GRANULE ALLOCATED BUT ASSIGNED TO MULTIPLE FILES

More than one file is using the same 5 sectors (GRANULE) to store its data. The last 'SAVE' or 'PUT' will have written to those five sectors and WRITTEN OVER the previous contents.

Determine which file was the LAST to use that granule. 'COPY' that file to another disk, then 'KILL' it on the original disk. 'LOAD' the remaining file, clean up the now garbled code, and 'SAVE' (or 'PUT') it back to that or another disk. Clean up any remaining 'GAT' errors by "ZAP"ing the 'GAT' table.

The number on the far left is the RELATIVE GRANULE in the 'GAT' TABLE. The number beside each file name is the 'DEC' of that file's entry in the directory sectors. The number beside the program name is the 'DEC' for that program. See figure 6.13 for details on decoding the 'DEC'.

3.4.5 GRANULE ALLOCATED BUT NOT ASSIGNED TO ANY FILE

A 'GRANULE' is not being used by any file. "ZAP" the offending GRANULE with the correct code. The number to the far left is

the RELATIVE GRANULE in the 'GAT' table. The number beside the program name is the 'DEC' for that program. See figure 6.13 for details on decoding the 'DEC'.

3.4.6 GRANULE LOCKED OUT, BUT FREE

A GRANULE has been LOCKED-OUT and may not be used by the system. "ZAP" the offending byte in the LOCK-OUT TABLE. The number to the far left is the relative GRANULE in the LOCK-OUT TABLE.

3.5 "LMOFFSET"

The real purpose of this program is to allow you to load and execute programs that 'normally' cannot be loaded with the DOS resident in RAM.

'LMOFFSET' first tells you where the program loads and entry point. Figure 3.4 is the prompt and output sequence of 'LMOFFSET'.

(figure 3.4)

```
APPARAT LOAD MODULE OFFSET PROGRAM, VERSION 1.1
SOURCE FROM DISK OR TAPE? REPLY "D" OR "T"? D
SOURCE FILESPEC?BASIC/CMD
MODULE LOADS TO 4D00-6431
MODULE OVERLAPS DOS RAM (4000-51FF)
MODULE LOAD WILL OVERLAP "CMD" PROGRAM AREA (5200-6FFF)
ENTRY POINT = 5BAD
NEW LOAD BASE ADDRESS (HEX)?
```

This program will tell you ABOUT the file; it will NOT tell you where it is on the disk or anything about the disk. It will assist you in locating a machine language program IN MEMORY so that it may be modified or corrections made to it prior to writing it back to disk.

It will also help in making a disassembly from the disk since you need to know the load address of the module before disassembling.

```
***** CAUTION *** CAUTION *** CAUTION *****
**
**      WHEN USING 'LMOFFSET' IN THIS MANNER,      **
**      DO NOT COMPLETE PROGRAM ORERATION --      **
**      If you complete the program's opera-      **
**      tion LMOFFSET will attach an 'APPEND-      **
**      AGE' to the program file causing it        **
**      to load in a place other than its         **
**      intended address!!!                        **
**
*****
```

If you are using NEW DOS, "J-K-L" the video display to your line printer (or make notes if you don't have a printer).

4.0 OPERATING SYSTEMS

This will be a brief review of the various operating systems that are available as of this writing. I will not dwell too long on the pros and cons of each and you must remember that the following is an OPINION, mine.

4.1 "TRSDOS 2.1"

Except for the few unfortunate souls that started with 2.0 this is the operating system that most of us developed our first, genuine love-hate relationship with. For all practical purposes, due to the short life of 2.0, this was the 'FIRST' operating system generally available for the TRS-80.

2.1 has many problems. Of course, Radio Shack never came out and admitted, in plain English, (at least to me - did they tell you?) that the problems existed. TRSDOS 2.1 is adequate for most trivial programming requirements and a few serious applications IF you are prepared to tolerate an occasional lost file. If you contemplate any real serious applications I would not recommend that TRSDOS 2.1 be used, under any circumstances.

Data recovery on TRSDOS 2.1 generated disks is normal and routine for formatted data disks and system disks.

4.2 "TRSDOS 2.2"

TRSDOS 2.2 is a huge improvement over 2.1. Most of the errors are corrected. However, it will still create errors. Most of the complaints I have about the system are that they still have not given the user any of the utility that is available with NEW DOS.

As far as data recovery goes, there is one major point. When you 'KILL' a file with 2.2, it ZEROS THE ENTIRE DIRECTORY ENTRY. There is not a single clue as to what was there or where it was! Since Radio Shack has no utility for looking at the disk, I presume it was to prevent all you "SUPERZAPPERS" out there from finding out too much! However, if you need to recover something, this makes it not impossible but a genuine bitch because you have to go 'mucking around on the disk' looking for the file.

For this reason alone, I would not use this system on a serious application where I MIGHT have to recover 'KILL'ed data.

Data recovery on TRSDOS 2.2 generated disks is normal and routine on formatted disks and system disks except for the above described 'KILL'ed files.

4.3 "VTOS 3.0"

This is Randy Cook's version of 2.2 with quite a few bells and whistles. Cook is the author of Radio Shack's 2.1 and, I have reason to suspect, most of 2.2. This system has some nice features but is, in my opinion, VERY AGGRAVATING to use because of its 'BACKUP' protection feature. In the version that I used for evaluation, some of the commands did not work entirely as advertised. I'm sure that this will be corrected in a later release. On the whole, the system is good and the concepts are excellent. I have not used it enough, at this time, to have detected any errors, if it has any.

If you find it necessary to recover data or files that have been 'SAVE'd to a VTOS 3.0 system disk, you will not be pleased with the recovery procedures

This is due to the fact that as a function of the VTOS 3.0 protection features, you will NOT BE ABLE TO RECOVER THE DATA TO ANOTHER DISK AND THEN 'RUN' THAT DISK!

In spite of all the nice features in this system, it is for this reason that I would not recommend its use with applications of other than, a trivial nature. Data recovery on VTOS 3.0 system disks is VERY DIFFICULT. You must first format a disk and then use the "SUPERZAP" 'BACKUP' function to transfer the information to the 'working disk'. YOU WILL NOT BE ABLE TO 'BACKUP' TRACK 0, SECTOR 4. You must 'SKIP' this sector when "SUPERZAP" tries to 'read' it from the 'SOURCE' disk. Then, when you have finished recovering the file, you must 'COPY' it back to a 'system disk' MADE FROM THE MASTER VTOS 3.0 YOU RECEIVED FROM MRS. COOK'S SON, RANDY.

VTOS 3.0 WILL NOT FUNCTION UNLESS TRACK 0, SECTOR 4 IS UNFORMATTED! (At least that's the way it appears.) This is how Randy Cook is able to protect his software from pirating. It is a great idea but it makes it extremely aggravating to use. For a new user who is trying to use an applications package transferred to this system, who is not familiar with computers, nor does he want to be -- he just wants to 'press a button and have the damn thing run his application -- this system will not find much favor at all.

4.4 "NEW DOS 2.1"

It works! The current release has no known bugs and will do everything Radio Shack says cannot be done. It corrects every KNOWN error in TRSDOS 2.1. All in all, there are over 200 additions, corrections, and enhancements to TRSDOS. Many of the 'improvements' in TRSDOS 2.2 are poor 'implementations' of NEWDOS 2.1. (That's an opinion, and I cannot verify it, but from the looks of things, I'd give better than even odds that it's true.)

NEWDOS 2.1 is oriented to the programmer as well as the user. Included in the NEW DOS+ package, are utilities such as "SUPERZAP", 'DIRCHECK', 'LMOFFSET' and others. These utilities are especially designed to assist the user and are very necessary if you need to recover data.

Data recovery on NEWDOS 2.1 generated disks is normal and routine for formatted data disks and system disks.

4.5 FUTURE OPERATING SYSTEMS.

The crystal ball business is tough. I have no reliable data on what Radio Shack's or Randy Cook's plans are for improved or new operating systems. I suspect that Radio Shack has had its attention diverted somewhat by trying to get out the new MODEL II unit and that the new unit will occupy much of their development time in the software area.

They will probably develop, at some future time, an operating system for the TRS-80 that emulates their larger machine.

This is only a guess, but I'll give odds, because they will want to use the MODEL II for internal development of all software. As a result they will have to devise ways of making some of the MODEL II features (whatever they are) available to the TRS-80 user. This will naturally lead to a system for the TRS-80 that emulates MODEL II.

Randy Cook is evidently no longer associated with RS and his company, Virtual Technology, Inc., will probably develop additional software for the TRS-80. It's my guess that VTOS 3.0 will go through several development stages that will range from corrections to improvements and finally enhancements. Cook is obviously very familiar with the TRS-80 and I would hazard a guess (AGAIN??) that he will continue to write software for the machine if only because he knows it so well.

I am very much in touch with Apparat so I do know some of the plans for their future TRS-80 developments. At this time NEW DOS is available in 35 and 40 track versions. A 77 track version of NEWDOS 2.1 will soon be forthcoming. This will be compatible with the Micropolis 77 track drives. My information is that these drives and the operating system will be available from APPARAT dealers in the early Fall of '79 if not sooner.

A 'SUPERDOS' is in work which will blow your socks off. I have had the opportunity to see some of its extended capabilities, especially in the file handling area, that will in my estimation, make the TRS-80 a viable business tool. It will also, so I'm told, be able to 'mix-and-match' disk drive units of 35, 40 and 77 tracks, ON THE SAME MODEL. Without going into a lot of detail, I'll just say that 'SUPERDOS' will be one light year ahead of anything you have seen so far --- BAR NONE!

```
***** WARNING ***** WARNING ***** WARNING ***** WARNING *****
**
** AS OF THIS WRITING (9/1/79) A NEW BUG HAS BEEN DIS-
** COVERED IN TRSDOS 2.2! (YES, THOSE WONDERFUL FOLKS
** IN FORT WORTH KNOW ABOUT IT - WHAT DID YOU EXPECT?
**
** IN ADDITION THERE IS A 2.3 VERSION OF TRSDOS AND IT
** IS BEING KEPT SECRET BY THE CRACK FORT WORTH SOFT-
** WARE DEVELOPMENT TEAM. (I HAVE THIS FROM A VERY RE-
** LIABLE SOURCE!)
**
** ----- BEWARE -----
**
** WHEN FILES ARE OPENED ON 2 SEPARATE DRIVES, WHILE IN
** BASIC AND ANY ONE FILE IS "CLOSED" THEN THE SPEC-
** IFIED 'CLOSE FILE' MAY BE 'KILLED'!!!! ALL SUBSE-
** QUENT 'CLOSES' ARE HANDLED CORRECTLY. THIS IS AN
** INTERMITTENT BUG AND MAY NOT FUNCTION EVERY TIME.
**
***** WARNING ***** WARNING ***** WARNING ***** WARNING *****
```

.... NEW DOS, Anyone?



5.0 DISK ORGANIZATION

In the TRS-DOS DISK OPERATING SYSTEM 2.1 MANUAL we are told that we have 67 GRANULES of free space on a formatted disk and somewhat less on a disk with a DOS. Here is a breakdown of the entire disk:

Tracks	
TRS DOS 2.1	35
NEW DOS 2.2	35 or 40
VTOS 3.0	35
SUPERDOS 1.0	18 to 80
NOTE:With SUPERDOS 1.0 you may mix and match disk drive units with different track configur- ations.	
Sectors per track	10
Sectors per Diskette	350 (35 track)
	400 (40 track)
	770 (77 track)
Sectors per 'GRANULE'	5
Bytes per Sector	256
Usable Bytes per Sector (TRSDOS 2.1) ..	255
(TRSDOS 2.2) ..	256
(NEWDOS 2.1) ..	255
(VTOS 3.0)	256
(SUPERDOS 1.0) ..	256
Bytes per Disk	89,600 (35 track)
	102,400 (40 track)
	197,120 (77 track)
Usable Bytes per Disk	85,410 (35 track)
Usable Sectors for Data Storage	335 (35 track)
GRANULES per Disk	70 (35 track)
Usable GRANULES per Formated Disk	67 (35 track)

A little simple math will verify the above figures. Each track, of which there are 35, has 10 sectors of 256 bytes per sector. That calculates out to 350 sectors per disk and 350 times 256 equals 89,600 bytes of storage.

The 'BOOT' and 'DIRECTORY' take 15 sectors of disk space. BOOT is physically located on track 0 and occupies sectors 0 through 4. DIRECTORY is located on track 11(HEX) <17 decimal> and occupies sectors 0 through 9.

TRS-DOS system programs use a large chunk of storage and leaves us with only 58,880 bytes of storage space on a disk with TRS-DOS. Radio Shack (in its infinite wisdom) decided to make it impossible to 'KILL' system files. (Corrected in TRSDOS 2.2.) As a result, the BASIC language programmer is cursed with what the manual nonchalantly describes as "...unexpected entry into DEBUG." In a few paragraphs you'll know now to remove the passwords and 'KILL' that damn (DE) BUG. Of course, if you are using NEW DOS+, you do not have this problem. Not only will you no longer have 'unexpected entry' after you 'KILL' DEBUG but, you'll have more disk space!

Back to business... Throughout this monograph, I will refer to the 'relative byte'. Imagine that the disk is composed of 350 blocks laid end to end. (See figure 5.1)

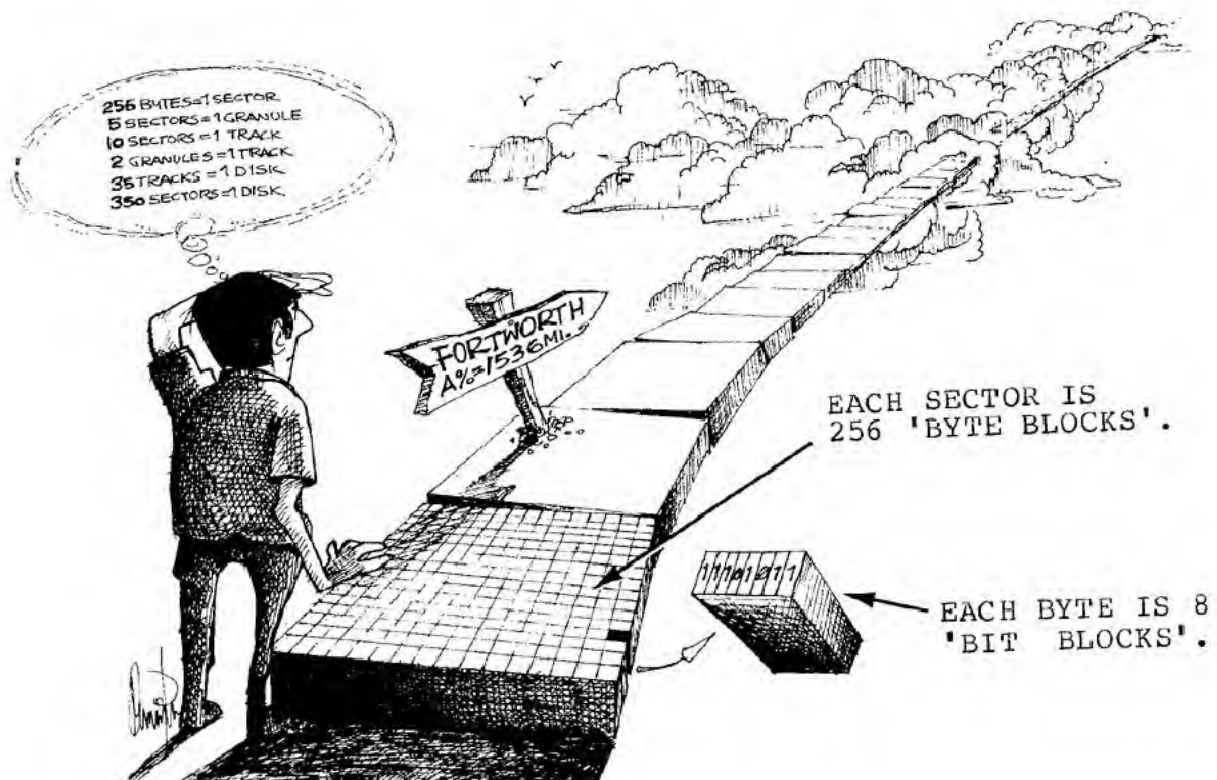
Every 10 blocks is a 'track'. Each block consists of 256 smaller blocks 16 across and 16 deep. The smaller blocks are the bytes. The first byte in the upper left hand corner is called the zeroth relative byte.

Counting across to the 16th byte-block gets us to the upper right hand corner of our 16 by 16 byte sector block and to the 15th RELATIVE BYTE. RELATIVE BYTE 16 begins on the first block of the second row down, and the 31st RELATIVE byte is the last block of the second row. This nonsense continues until we get to the lower right hand corner of our sector and we are at the 255th RELATIVE byte.

To compound the matter even further, each byte 'block' is made of 8 smaller 'blocks'. These 'blocks' are the BITS. Each bit can store only one of two values. A '1' or a '0'. I won't go into bits here and suggest Bardens' book for a very thorough discussion on the subject of bits and binary arithmetic.

(figure 5.1)

VISUALIZING SECTORS AS 350
BLOCKS LAYED END TO END.



Imagine that every 10 of the big sector blocks is a TRACK. Each track is numbered from ZERO to twenty-two (HEX). Some of these tracks are 'dedicated' to particular SYSTEM PROGRAMS. Figure 5.2 is a 'DISK MAP' of the tracks and the space 'dedicated' to certain programs. All programs with the filename extension of '/SYS' are programs of this nature. 'FORMAT/CMD', BASIC/CMD, 'BACKUP/CMD' are certainly important to the system but it is not necessary for them to be in any particular place on the disk.

Actually there are only a few areas on the disk that MUST CONTAIN SPECIFIC OBJECT CODE MATERIAL. These are 'BOOT/SYS', 'SYS0/SYS' and 'DIR/SYS'. 'BOOT' must always be located on Track 0, beginning at sector zero. 'SYS0/SYS' must be located on track '0', sector 5 and 'DIR/SYS' must be located on Track 11 (HEX) beginning at sector zero.

The directory may be moved (it's a hassle) to another location. It MUST also be read protected. If the directory is moved, 'SAVE' has a bitch of a time trying to figure out where to put the directory information since it expects the directory to be on track 11 (HEX). Eventually it will find it and deposit its data in the right places. This can be speeded up a bit by changing relative byte '02'(HEX), in the 'BOOT' (track '0', sector '0') to the HEX value of the track you have moved the directory to.

The 'BOOT' is not actually a program but rather a machine language 'TABLE' that is automatically loaded on power-up or reset --- sometimes referred to as 'IPL'. (Initial Program Load. 'IPL' is computer jargon for, "Push the button, Hilda!")

Figure 5.2 is a 'MAP' of a typical 'SYSTEM DISK' (TRS-DOS 2.1). You will notice that the system programs are grouped together. It is not absolutely necessary that this always be the case. In fact it is possible to put the SYSTEM programs anywhere except for 'BOOT/SYS', 'SYS0/SYS' and 'DIR/SYS'. NEWDOS requires that 'SYS13/SYS', when it is resident on the diskette, to be specifically located also.

Other programs such as FORMAT/CMD and BASIC/CMD may not be in the same location on your disk, especially if you have 'COPY'ed these programs from another disk.

Disk allocation is handled in groups of 5 sectors at a time. (More on this in chapter 6.) For this reason every program or file is allocated disk space in 5 sector chunks called "GRANULES".

TRS-DOS 2.1 and 2.2 assign a MINIMUM of two GRANULES at a time. That is why you run out of disk space so quickly when you have a bunch of small files or programs. NEW DOS assigns only one GRANULE at a time.

You can test this by saving a one line BASIC program to disk. Before you save the program run "SUPERZAP" and look at the 'GAT' sector's GRANULE allocation. 'SAVE' the program then look at the 'GAT' sector again. Chapter 6 will explain the meaning of the 'GAT' sector so you will be able to interpret the results.

TRS DOS 2.1 DISK MAP (35 TRACK)

TRACK		GRANULE		TRACK		CONTENTS	
NUMBER		NUMBER		SECTORS 0 - 4		SECTORS 5 - 9	
HEX/DECIMAL		(HEX)		: <-----GRANULE----->:		: <-----GRANULE----->:	
0	- 0	0	& 1	:	<-----BOOT/SYS----	:	<-----SYS0/SYS----
1	- 1	2	& 3	:	<-----SYS0/SYS----	:	<-----SYS0/SYS----
2	- 2	4	& 5	:	<---FORMAT/CMD---	:	<---FORMAT/CMD---
3	- 3	6	& 7	:	<---FORMAT/CMD---	:	<---BACKUP/CMD---
4	- 4	8	& 9	:	<---BACKUP/CMD---	:	<---BACKUP/CMD---
5	- 5	A	& B	:	<-----FREE----->	:	<-----FREE----->
6	- 6	C	& D	:	<-----FREE----->	:	<-----FREE----->
7	- 7	E	& F	:	<-----FREE----->	:	<-----FREE----->
8	- 8	10	& 11	:	<-----FREE----->	:	<-----FREE----->
9	- 9	12	& 13	:	<-----FREE----->	:	<-----FREE----->
A	- 10	14	& 15	:	<-----FREE----->	:	<-----FREE----->
B	- 11	16	& 17	:	<-----FREE----->	:	<-----FREE----->
C	- 12	18	& 19	:	<-----FREE----->	:	<-----FREE----->
D	- 13	1A	& 1B	:	<-----FREE----->	:	<-----FREE----->
E	- 14	1C	& 1D	:	<-----FREE----->	:	<-----FREE----->
F	- 15	1E	& 1F	:	<-----FREE----->	:	<-----FREE----->
10	- 16	20	& 21	:	<-----SYS1/SYS----	:	<-----SYS2/SYS----
11	- 17	22	& 23	:	<-----DIR/SYS----	:	<-----DIR/SYS----
12	- 18	24	& 25	:	<-----SYS3/SYS----	:	<-----SYS4/SYS----
13	- 19	26	& 27	:	<-----SYS5/SYS----	:	<-----SYS6/SYS----
14	- 20	28	& 29	:	<-----SYS6/SYS----	:	<-----SYS6/SYS----
15	- 21	2A	& 2B	:	<---BASIC/CMD---	:	<---BASIC/CMD---
16	- 22	2C	& 2D	:	<---BASIC/CMD---	:	<---BASIC/CMD---
17	- 23	2E	& 2F	:	<-----FREE----->	:	<-----FREE----->
18	- 24	30	& 31	:	<-----FREE----->	:	<-----FREE----->
19	- 25	32	& 33	:	<-----FREE----->	:	<-----FREE----->
1A	- 26	34	& 35	:	<-----FREE----->	:	<-----FREE----->
1B	- 27	36	& 37	:	<-----FREE----->	:	<-----FREE----->
1C	- 28	38	& 39	:	<-----FREE----->	:	<-----FREE----->
1D	- 29	3A	& 3B	:	<-----FREE----->	:	<-----FREE----->
1E	- 30	3C	& 3D	:	<-----FREE----->	:	<-----FREE----->
1F	- 31	3E	& 3F	:	<-----FREE----->	:	<-----FREE----->
20	- 32	40	& 41	:	<-----FREE----->	:	<-----FREE----->
21	- 33	42	& 43	:	<-----FREE----->	:	<-----FREE----->
22	- 34	44	& 45	:	<-----FREE----->	:	<-----FREE----->

6.0 THE DIRECTORY

The key to finding anything on the disk is the directory. Even the operating system can't find anything without the directory. Now that we have a basic understanding of how the disk is organized, we'll take a very close look at the directory. I'll explain what each byte means, what it does, and how to use it to find things just as the operating system does.

The directory is located on track 17 (11 HEX). It is composed of 10 sectors of 256 bytes per sector. This gives the directory 2,560 bytes in which to store data. There are no unused bytes in the directory. Figure 6.3 is a 'MAP' of the DIRECTORY.

The minimum space allocated for storing any type of file, is one "GRANULE". (No, Virginia, I do not know where the word "GRANULE" came from. Perhaps it describes the size brain of the person who thought of inventing another 'computer jargon' term.) At any rate, the over-all scheme for representing free space is as follows:

5 sectors = 1 granule 2 granules = 1 track

When you do a 'FREE', GRANULES are shortened to 'GRANS' --- it will look like figure 6.1.

(figure 6.1)

DRIVE 0 --	TRSDOS	11/27/78	41 files	42 GRANS
DRIVE 1 --	TRSDOS	01/01/79	33 files	6 GRANS

With that out of the way let's dive into the directory. (Appendix A contains a 'DIRECTORY TRACK DUMP' of TRSDOS, NEWDOS, and VTOS. made by using the 'PD' - 'PRINT DISK SECTORS' function of "SUPERZAP").

We will discuss each sector and then each entry in each sector.

"GAT SECTOR" - SECTOR 0

(figure 6.2)

011000	FFFC	FCFC	FCFC	FCFF	FEFE	FCFD	FCFC	FCFC
011010	FCFF	FCFC	FFFC	FEFD	FCFD	FDFC	FCFC	FEFC
011020	FCFC	FCFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
011030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
011040	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
011050	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
011060	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC
011070	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC
011080	FCFC	FCFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
011090	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0110A0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0110B0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0110C0	FFFF	FFFF	FFFF	FFFF	FFFF	FF21	0000	E042!...B
0110D0	5452	5344	4F53	2020	3034	2F30	312F	3739	NOTES...04/01/79
0110E0	0D0D	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
0110F06	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF

DIRECTORY TRACK MAP

TRACK 17 (11 HEX)

SECTOR
NAME & NUMBERSECTOR
CONTENTS

GAT	0		Unassigned granules Assigned granules Locked-out granules Master disk password. Disk name & date 'AUTO' command file
HIT	1		Program name 'hash code' 'DEC' of 'FPDE-FXDE'
FPDE/FXDE	2		Type of file entry (FPDE - FXDE) File type Entry status Space availability status 'EOF' Logical record length (NOT USED BY BASIC) File name File name extension Update password Access password Number of sectors assigned to file File extents Track location Sector location in track Number of contiguous sectors in extent Entry type (FPDE - FXDE)
		1	
FPDE/FXDE	3		
		2	
FPDE/FXDE	4		
		3	
FPDE/FXDE	5		
		4	
FPDE/FXDE	6		
		5	
FPDE/FXDE	7		
		6	
FPDE/FXDE	8		
		7	
FPDE/FXDE	9		
		8	

The actual directory entries are located in these eight sectors.

These numbers correspond to the vertical columns beginning at relative byte '00 - 0F'. See 'HIT MAP' figure 6.8.

"GAT" stands for GRANULE ALLOCATION TABLE. This sector contains all of the information the DOS needs to allocate space for files. It also is the sector that notes 'lockout' on tracks.

Figure 6.6 is a 'MAP' of the "GAT" sector with an explanation of the various GAT areas.

You will notice in figure 6.2 that the first 35 bytes contain 'FF's, 'FC's and 'FE's. These first 35 bytes represent the 35 tracks. An 'FF' in one of these bytes means that the track is full. An 'FE' means the first 5 sectors are available and an 'FD' means the last 5 sectors are available. Also see figure 6.4, below.

At relative byte '60' (figure 6.2) you will notice a replay of the first 3 lines of this sector. This is where 'TRACK LOCKED OUT' information is maintained. Beginning at relative byte '60' you will see 35 'FC's. This means that all 35 tracks are available to the system. If there is an 'FF' in one of these 35 bytes AND a corresponding 'FF' in the above set of 35 bytes, a track has been "LOCKED-OUT".

At relative byte 'CB' and for the next 3 bytes, there is a '21 0000'. What ever these codes are they are not used by the system.

Relative bytes CE and CF are the 'hash code' for the master disk password.

The next line (beginning at relative byte 'D0'), contains the disk name and the backup date.

Bytes 'E0' to 'EF' and 'F0' to 'FF' are the 'command file' for the 'AUTO' function. These 32 bytes will contain the name of any program, and/or command that has been defined as 'AUTO' while in DOS. If byte 'E0' contains a '0D' (carriage return) then the 'AUTO' function will not execute.

(figure 6.4)

BINARY	HEX	MEANING
11111111	FF	1st & 2nd granules allocated. (sectors 0 - 9)
11111110	FE	2nd granule allocated (sectors 5 - 9)
11111101	FD	1st granule allocated (sectors 0 - 4)
11111100	FC	1st & 2nd granules free (sectors 0 - 9)

"HIT SECTOR" - SECTOR 1

"HIT" stands for HASH INDEX TABLE. This sector contains a 'HASH CODE' that relates to each stored FILE NAME. The location of the hash code also tells the DOS where the file information is on the directory. Figure 6.5 is a dump of a 'HIT' sector and figure 6.8 is a MAP of the 'HIT' sector.

There is a one byte hash code for each program or file stored. The position as well as the code is important.

A hash code is a number that is derived by some scheme of assigning each letter a numerical value. Then, depending on each letters position, they are multiplied by some number and the result of each multiplication is added then divided and rounded. Eventually a code number results which is a 'HASH' of the original entry.

There are literally millions of schemes for hashing and the hash code could be any number of bytes long depending on who is using it and for what.

In this particular case the HASH code is 1 byte. There will be more on 'HASH CODES' in the data recovery chapter.

(N 2 3 4 5 6 7 8) (figure 6.5)

HIT	SECTOR	1	2	3	4	5	6	7	8
011100	A22C	2E2F	2C2D	2A2B	0000	0000	0000	0000	.,./,*+.....
011110	0000	0000	0000	0000	0000	0000	0000	0000	(.....&.....
011120	2800	0000	00A7	26A6	0000	0000	0000	0000
011130	0000	0000	0000	0000	0000	0000	0000	0000
011140	F200	8900	0000	0000	0000	0000	0000	0000
011150	0000	0000	0000	0000	0000	0000	0000	0000
011160	0000	5600	00C5	0000	0000	0000	0000	0000
011170	0000	0000	0000	0000	0000	0000	0000	0000
011180	7900	AD00	0032	0000	0000	0000	0000	0000	.U.....
011190	0000	0000	0000	0000	0000	0000	0000	0000
0111A0	F01D	8F00	009D	00B7	0000	0000	0000	0000F2.....
0111B0	0000	0000	0000	0000	0000	0000	0000	0000
0111C0	0067	0000	0000	0000	0000	0000	0000	0000
0111D0	0000	0000	0000	0000	0000	0000	0000	0000
0111E0	A3DB	0000	0000	00EE	0000	0000	0000	0000
0111F06	0000	0000	0000	0000	0000	0000	0000	0000

(Columns 1 2 3 4 5 6 7 8)

At the bottom of the 'HIT' MAP there are columns numbered 1 through 8. Each of these eight VERTICAL columns represent the eight sectors available for storing file names and each even numbered row across is the relative byte the entry starts on in its sector.

For instance, in figure 6.5 there is a hash code at relative byte 'A2'. (The hash code I'm referring to is '8F')

The 'FPDE' or FILE ENTRY is in VERTICAL Column 3. This means that the file that corresponds to this hash code is in the third sector AFTER THE 'HIT' SECTOR, which is relative sector 4. (Also see the DIRECTORY TRACK MAP figure 6.3.)

Also notice that the hash code 'A2' is in vertical column 1 and at relative byte '00'. This 'points' or corresponds to the first sector after the 'HIT' sector (relative sector 2) and relative byte '00' in that sector. This is the hash for 'BOOT/SYS'. Next to 'A2' is the hash code '2C' and this points to the second sector after the 'HIT' sector and also points to relative byte '00'. This is the hash for 'DIR/SYS'.

Also notice that there are two '2C's on that line. One is for 'DIR/SYS' and the other is for 'SYS2/SYS' and the only conclusion is that the hash codes need not be unique but must be derived from the program name and be in the correct corresponding byte which points to the 'FPDE' entry in that sector.

GAT SECTOR (figure 6.6)
 'GAT' SECTOR MAP (TRACK 11, SECTOR 0) 35 TRACK DOS

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	← GRANULE ALLOCATION TABLE →															
10																
20																
30																
40																
50																
60	← TRACK LOCK OUT TABLE →															
70																
80																
90																
A0																
B0																
C0												← UNKNOWN →		← PSW →		
D0	← DISK NAME AND DATE →															
E0	← 'AUTO' COMMAND FILE →															
F0																

The following 'GRANULE ALLOCATION MAP' is a detail of figure 6.6. It is also extended to include the GRANULES to track 80 if you should ever have a system that uses this many tracks.

(figure 6.7)

'GRANULE ALLOCATION MAP'

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16
	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
10	00	02	04	06	08	0A	0C	0E	10	12	14	16	18	1A	1C	1E
	01	03	05	07	09	0B	0D	0F	11	13	15	17	19	1B	1D	1F
20	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
30	20	22	24	26	28	2A	2C	2E	30	32	34	36	38	3A	3C	3E
	21	23	25	27	29	2B	2D	2F	31	33	35	37	39	3B	3D	3F
40	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
	22	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
50	40	42	44	46	48	4A	4C	4E	50	52	54	56	58	5A	5C	5E
	41	43	45	47	49	4B	4D	4F	51	53	55	57	59	5B	5D	5F
60	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64
	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
70	60	62	64	66	68	6A	6C	6E	70	72	74	76	78	7A	7C	7E
	61	63	65	67	69	6B	6D	6F	71	73	75	77	79	7B	7D	7F
80	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80
	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
90	80	82	84	86	88	8A	8C	8E	90	92	94	96	98	9A	9C	9E
	81	83	85	87	89	8B	8D	8F	91	93	95	97	99	9B	9D	9F

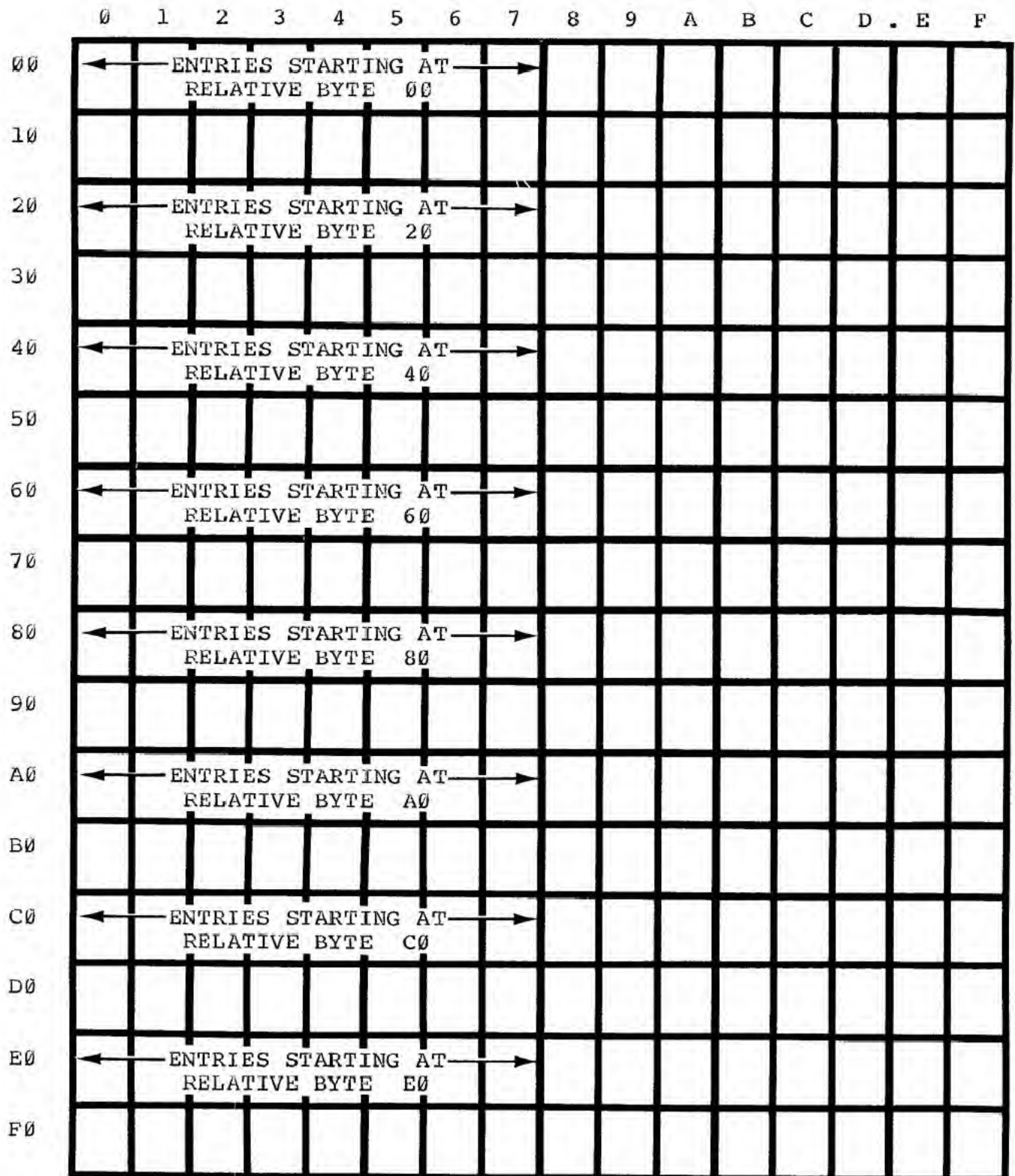
LEGEND

01	<-- Track (DECIMAL)
00	<-- Track (HEX)
00	<-- 1st GRANULE (HEX)
01	<-- 2nd GRANULE (HEX) (In this track)

GRANULE ALLOCATION CODE

FF	= 1st & 2nd GRANULES allocated
FC	= 1st & 2nd GRANULES free
FD	= 1st GRANULE allocated
FE	= 2nd GRANULE allocated

(figure 6.8)
 'HIT' SECTOR MAP (TRACK 11, SECTOR 1)



1 2 3 4 5 6 7 8 ← These numbers are the first eight vertical columns of this map and they represent the 8 sectors used for actual directory entries. Also see figure 6.3

'FPDE/FXDE SECTORS SECTORS 2 - 9

'FPDE' stands for FILE PRIMARY DIRECTORY ENTRY and 'FXDE' is defined as 'FILE EXTENSION DIRECTORY ENTRY.' These sectors are the actual directory. (Also see figure 6.10, FPDE/FXDE SECTOR MAP.)

The program name, attributes, passwords, size (in sectors) 'END OF FILE', and physical location on the disk are stored here.

(figure 6.9)

```

011400 5F00 0000 0053 5953 3020 2020 2053 5953 .....SYS0....SYS
011410 EB29 210E 0F00 0022 FFFF FFFF FFFF FFFF .)!....".....
011420 0000 0000 0000 0000 0000 0000 0000 0000 .....
011430 0000 0000 0000 0000 0000 0000 0000 0000 .....
011440 1000 009A 0045 4454 4153 4D20 2043 4D44 .....EDTASM..CMD
011450 9642 9642 2000 0D24 1A01 FFFF FFFF FFFF .B.B...$......
011460 1000 00B7 0054 5253 3233 3220 2020 2020 .....TRS232.....
011470 9642 9642 0300 1D20 FFFF FFFF FFFF FFFF .B.B.....
011480 0000 0000 0000 0000 0000 0000 0000 0000 .....
011490 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114A0 0000 00D9 0054 5249 4254 5241 5020 2020 .....TRIBTRAP...
0114B0 9642 9642 2100 1E22 2023 FFFF FFFF FFFF .B.B!..."#. ....
0114C0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114E0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0114F06 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

In addition to the 'FPDE' the 'FXDE's are also stored here. When there is not enough room to store all the information DOS needs about a file, it creates a 32 byte extension to the original 32 byte 'FPDE'.

Perhaps this would also be a good time to define a 'FILE'. A BASIC program stored with a 'SAVE' is a 'file'. A machine language or assembler program stored by using 'DUMP', 'TAPEDISK' or 'EDTASM' is a file. Data stored by using the 'OPEN' statement in BASIC is a file. In fact, anything that gets put onto the disk, with a name, is a file. (Is there any more confusion about 'files'? Good.)

Each directory sector, beginning at relative sector 2, may contain up to eight file names. Each of these file "ENTRIES" occupies 32 bytes. The first entry of each of these eight sectors is reserved for SYSTEM FILES.

You will note that each entry starts at one of the following RELATIVE bytes. (Also see figure 6.10.):

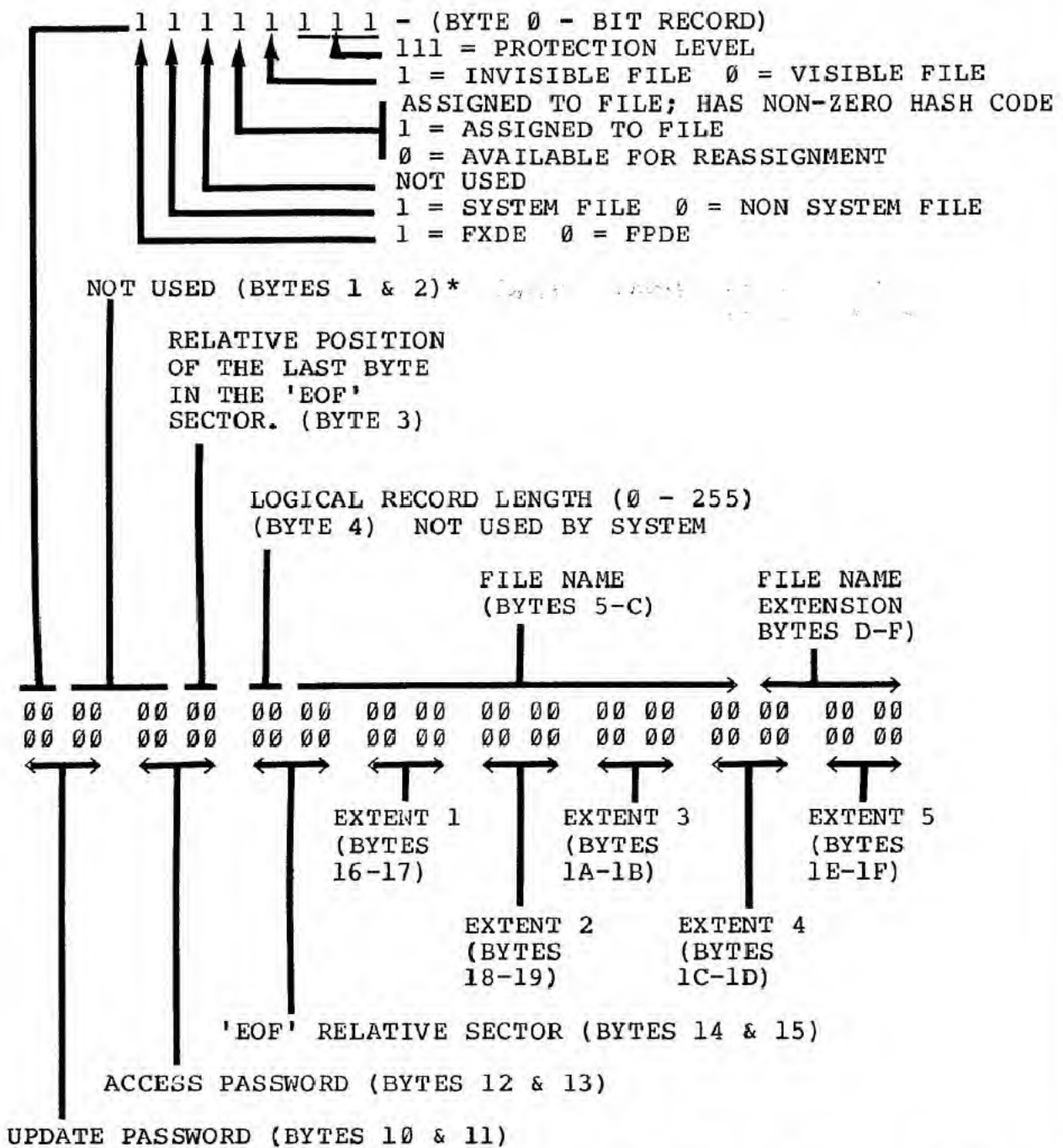
00, 20, 40, 60, 80, A0, C0, and E0.

Now let's examine a directory entry in detail. We will take the first thirty-two byte 'FPDE' entry of sector 2 and take it apart. (Figure 6.11.)

(figure 6.10)
'FPDE' - 'FXDE' SECTOR MAP (TRACK 11, SECTORS 2 - 9)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00	← 00	01	02	03	04	— DIRECTORY ENTRY ONE —										0F
10	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	→ 1F
20	← 20	21	22	23	24	— DIRECTORY ENTRY TWO —										2F
30	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	→ 3F
40	← 40	41	42	43	44	— DIRECTORY ENTRY THREE —										4F
50	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	→ 5F
60	← 60	61	62	63	64	— DIRECTORY ENTRY FOUR —										6F
70	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	→ 7F
80	← 80	81	82	83	84	— DIRECTORY ENTRY FIVE —										8F
90	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	→ 9F
A0	← A0	A1	A2	A3	A4	— DIRECTORY ENTRY SIX —										AF
B0	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	→ BF
C0	← C0	C1	C2	C3	C4	— DIRECTORY ENTRY SEVEN —										CF
D0	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	→ DF
E0	← E0	E1	E2	E3	E4	— DIRECTORY ENTRY EIGHT —										EF
F0	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	→ FF

'FPDE' DIRECTORY ENTRY

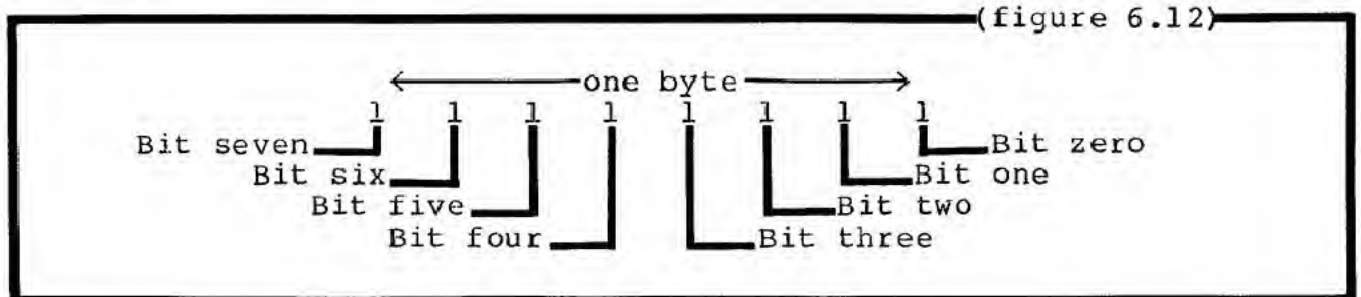


* NOTE: BYTE 1 is used by an 'FXDE' entry as a 'DEC' pointing to the original 'FPDE' entry.

Now we'll take each item in order. Notice that in figure 6.11, each byte is named with its hexadecimal number.

'FPDE' BYTE 0

FILE TYPE - SYS/NON-SYS FILE - ATTRIBUTES - ASSIGNMENT STATUS -PROTECTION LEVEL. The first byte is a bit record. The right most 5 bits (0 - 4) are considered separately and the left most 3 bits (5 - 7) are considered as one unit. Figure 6.12 illustrates the method for counting bits.



(The following is a review of the material in figure 6.11)

Bit 7 ... 1 = FXDE 0 = FPDE
if this bit is 'OFF' (Therefore a '0') then this entry is a 'PRIMARY' entry. If it is a '1' then the entry is an 'EXTENSION' of another entry location somewhere in the directory. There will be no hash code corresponding to an 'FXDE' entry.

Bit 6 ... 1 = The entry is a 'SYSTEM' file. A typical system file would be 'SYS3/SYS' and for that file this bit would be a one.

Bit 5 ... Not used.

Bit 4 ... 1 = Assigned to a file and has a non-zero hash code.
0 = This directory space is available for reassignment.

Bit 3 ... 1 = INVISIBLE FILE 0 = VISIBLE FILE An example of an INVISIBLE FILE is 'BASIC/CMD' on every TRS-DOS 2.1 or 2.2.

Bit 2-1-0 ... These 3 bits are to be interpreted as a unit; i.e., 111 binary = 7 decimal

BINARY	DECIMAL	PROTECTION LEVEL
111	= 7	= No access
110	= 6	= Execute only
101	= 5	= Read/execute
100	= 4	= Write/read/execute
011	= 3	= NOT USED
010	= 2	= Rename/write/read/execute
001	= 1	= Kill/rename/write/read/execute
000	= 0	= No restrictions

'FPDE' BYTES 1 & 2

If the entry is an 'FPDE' entry then these are not used and always contain zeros. If the entry is an 'FXDE' then BYTE '1' is the 'DEC' pointing BACK to the 'FPDE'. Byte '2' is never used and always contains '00'.

'FPDE' BYTE 3

END OF FILE (EOF) BYTE. This byte is the relative byte position of the last byte (of the file) in the last relative sector of the file. If you had a file that was 4 sectors long and this byte was a '13' then your file would end AT relative byte 13 (HEX) in sector 4.

'FPDE' BYTE 4

LOGICAL RECORD LENGTH. This neat idea is not used by the system! Evidently it's another good idea that was not 'implemented'. I suspect that it was to be used by the random file statements to make computing logical record lengths easier. In any case, this byte should be '00' but you can use it for anything you want seeing as how it isn't used.

'FPDE' BYTES 5 - C

FILE NAME. These eight bytes are the file name. The '/' is not stored. You may change or swap file names using "SUPERZAP" but be sure and change the hash code.

'FPDE' BYTES D - F

FILE NAME EXTENSION. Here is where the '/BAS' and other file name extensions are stored. These may be "ZAP"ped also. The extension is used in computing the hash code for 'HIT' sector so you will need to put in a proper hash code if you change the extension.

'FPDE' BYTES 10 & 11

UPDATE PASSWORD. Finally, the passwords. (Calm down, I'll explain how to unlock the passwords in a couple of chapters.) The UPDATE PASSWORD is a two byte hash code of the password you specify when you use the DOS command 'ATTRIB'. (See "TRS-DOS & DISK BASIC Reference Manual" section 4 page 12 for a complete (if obscure) explanation of 'ATTRIB')

'FPDE' BYTES 12 & 13

ACCESS PASSWORD. This is also a two byte hash code. This password is created when you specify a filespec thus:

SAVE"RSSALES/PSN.DUMB

In this case the material to the right of the '.' will be hashed and inserted into bytes 12 and 13. You may also change, delete and specify the ACCESS password with the DOS command 'ATTRIB'.

'FPDE' BYTES 14 & 15

END OF FILE (EOF) RELATIVE SECTOR. This is a tricky one. The concept is simple and straightforward; these bytes contain a count of the number (in HEX, of course) of sectors in the file. There are however, two sets of rules governing the use of these bytes:

DUMB RULE # 1 - If the 'EOF' byte contains '00' (in this case '00'=256 DEC) then this byte will contain the actual RELATIVE sector count.

DUMB RULE # 2 - If the 'EOF' byte contains any value OTHER THAN '00' then this byte will contain the RELATIVE sector count plus one!

Let's see how that works again. Suppose we have a short file that is EXACTLY 256 bytes long and we save it to disk. Now that will fit into one sector of storage and all of the file will be contained in relative sector 0 of the file. In this case 'FPDE' BYTES 14 & 15 will contain '01'. Now that makes sense! In all of the other 'counts' we make, we start counting with zero (I'll admit that it's kind of hard to get used to at first, but it IS a logical concept) and here we have a file stored in the 'zeroeth' sector and we have '01' stored in the relative sector count of the 'FPDE'.

Now let's take a little longer file - say, one about 600 bytes long. This file will require a little over 2 sectors of space. This means that the file will end in the second relative sector. (Counting from zero that's: 0, 1, 2.) In other words it takes 3 sectors to save it but, using our 'normal' count method the file will be in RELATIVE sector 2. NOW - using DUMB RULE # 2 - an '03' will be stored in the EOF SECTOR BYTE! (RELATIVE SECTOR COUNT = 2. EOF SECTOR BYTE = 2 + 1)

Jeez! you'd've thought they could at least be consistent. Oh well, you MUST realize that the folks that thought this up are the same wonderful folks that brought you the 400 name / 7 hour sort / MAIL LIST program and the 'monthly' newsletter that was published 5 times in two years.

Hold it! We're not through yet. We have one more thing to get straight and that's REALLY large files. Let's do a little more 'supposing'. Suppose you had a data file that occupied AN ENTIRE DISK. That would be 335 sectors. The largest number we can fit into a byte is 'FF' (HEX) and that equals 255 (DECIMAL). Now, even a Radio Shack store manager can figure out that we need more than one byte to store a gigantic number like 335.

Here, finally, is an example of such an 'EOF' sector byte:

335 (DECIMAL) = 014F (HEX)
EOF sector byte = 4F01

It's fairly obvious that the numbers are simply 'back-to-front' and all you have to do is put the '01' in front of the '4F' and you have it! Convert the number back to decimal and you'll know the number of sectors in this file.

'FPDE' BYTES 16-17, 18-19, 1A-1B, 1C-1D, 1E-1F

EXTENT 1, EXTENT 2, EXTENT 3, EXTENT 4, EXTENT 5. The EXTENTS contain the TRACK, GRANULE off-set, number of CONTIGUOUS granules (in the extent) and when necessary, the 'FXDE' pointer.

By now your lightning-quick-bear-trap-mind should be working at peak efficiency so I expect that you'll have no trouble understanding the EXTENTS.

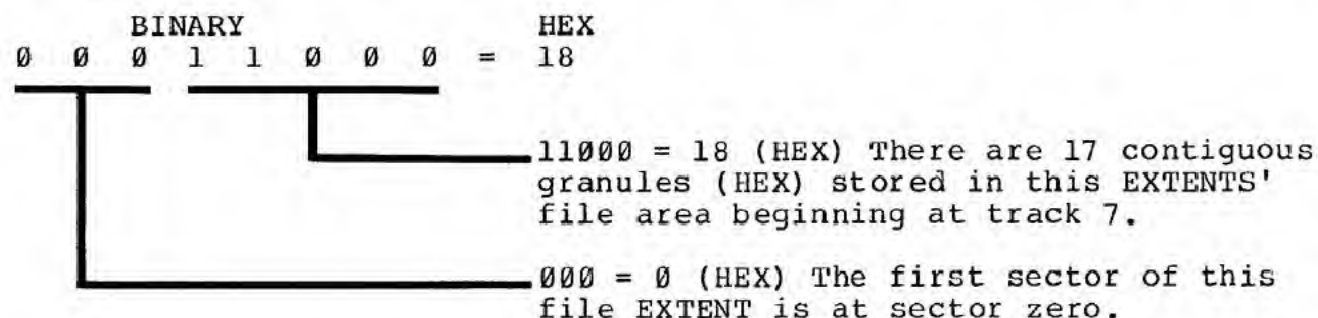
So far, we have all the information on a file we need to determine its name, length, and so on, but we still don't know exactly WHERE it is on the disk. This information is recorded in the EXTENT elements.

Consider the following EXTENT: 0718

The first byte of an EXTENT is the TRACK number in hexadecimal. In this example TRACK = 07

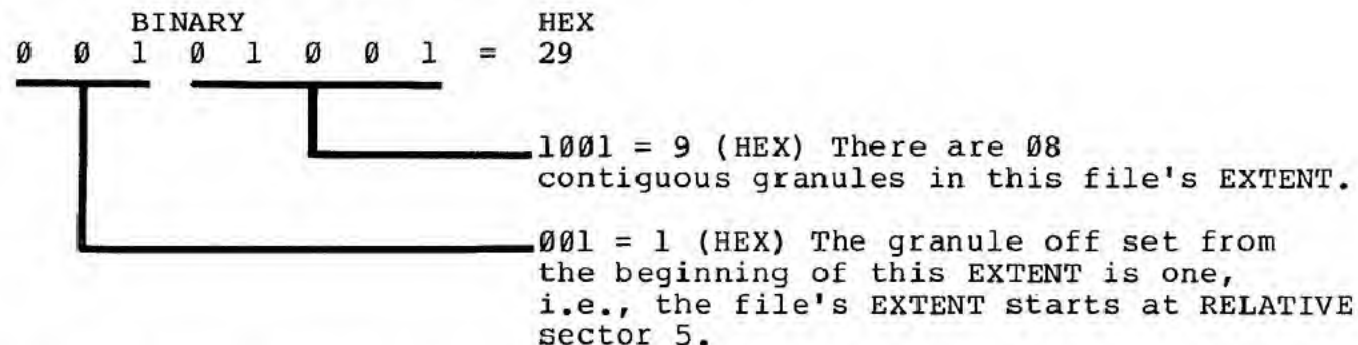
The second byte is a bit record. The right most 5 bits are the NUMBER OF CONTIGUOUS GRANULES ASSIGNED TO THIS EXTENT LESS ONE! The left most 3 bits is the OFF SET OF THE START OF THE FILE, FROM THE DESIGNATED TRACK, SECTOR ZERO, TO THE START OF THE FILE IN GRANULES (1=1 GRANULE 0=0 GRANULES).

Here is the bit record for the second byte of the above EXTENT example:



Here is another typical EXTENT we can decode: 1A29

TRACK = 1A (THAT'S EASY!)



From this we may surmise the following:

- (1) The track is easy; just read it.
- (2) If the second byte of the EXTENT is 19 or less then the file begins at SECTOR 0.
If the second byte of the EXTENT is 20 or greater then the file begins at SECTOR 5.

'FPDE' END OF EXTENTS.

All this is just fine, you say, but what in the dirty hell are all those 'FFFF's at the end of the EXTENTS? Just that, my fine feathered friend, the END OF THE EXTENTS. 'FFFF' means that there are no more EXTENTS. If you add to your file, and DOS cannot continue to add to an existing disk file area, then it will find some open (FREE) space, using the GAT table and then put the file in the newly allocated granules and construct a new EXTENT.

A file may have up to FIVE extents in a 'FPDE' and that brings us to....

'FXDE' ENTRIES

The mysterious 'FXDE' is about to be unmasked. If an 'FXDE' exists, you will see, IN EXTENT 5, of the 'FPDE', 'FE' followed by the 'DEC' of the 'FXDE'. Figure 6.14 is a typical example of a 'DEC' pointing to an 'FXDE'. Whats a 'DEC', you ask? Pay attention because there will be a test on this tomorrow.

'DEC' is defined as: DIRECTORY ENTRY CODE. The example in figure 6.13 will further your understanding so please press on ...

(figure 6.13)
32 byte 'FPDE' entry showing a 'DEC' ('pointer') to the 'FXDE' entry.

```
0119C0 1000 002B 0044 4953 4B4F 5247 2050 434C ...+.DISKORG.PCL
0194D0 9642 9642 4200 2023 0124 0500 0701 FE40 .B.BB..#.$.....@
```

These 2 bytes in EXTENT 5
'point' to the 'FXDE'.

FE (HEX) SIGNIFIES THAT THE NEXT BYTE IS THE 'DEC' TO
AN 'FXDE'

40 (HEX) = 01000000 (BINARY)

0 1 0 0 0 0 0 0 (BINARY)

000 = 000 (HEX) + 2 = 2. This is
the relative directory sector
the 'FXDE' is located in.

00 - Not used.

010 = 2 (HEX) This is the relative
32 byte directory entry in
that sector. REMEMBER TO
START COUNTING FROM ZERO!
(i.e., "0 - 1 - 2")

Next, let's look at the actual 'FXDE' entry in relative sector 2. See figure 6.14 below.

(figure 6.14)

```

011200 5E00 0000 0042 4F4F 5420 2020 2053 5953 .....BOOT....SYS
011210 EB29 210E 0500 0000 FFFF FFFF FFFF FFFF .)!.....
011220 0000 0000 0000 0000 0000 0000 0000 0000 .....
011230 0000 0000 0000 0000 0000 0000 0000 0000 .....
011240 90C7 0000 0000 0000 0000 0000 0000 0000 .....
011250 0000 0000 0000 0821 FFFF FFFF FFFF FFFF .....
011260 0000 0000 0000 0000 0000 0000 0000 0000 .....
011270 0000 0000 0000 0000 0000 0000 0000 0000 .....
011280 0000 0000 0000 0000 0000 0000 0000 0000 .....
011290 0000 0000 0000 0000 0000 0000 0000 0000 .....
0112A0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0112B0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0112C0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0112D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0112E0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0112F06 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

DIRECTORY SECTOR with 'FXDE' at relative byte '40'.

The first byte of the 'FXDE' is decoded in exactly the same manner as an 'FPDE'. The 'C7', at relative byte '41' is the 'DEC' that points BACK to the 'FPDE'.

It has been suggested by Fenwyler T. Murphy, a nephew of THE Murphy, that this may also be a seed value for generating a random error during a 'WRITE' operation that will invoke the TRS-DOS hidden command, 'DESDSK' (DESTROY DISK).

Aw c'mon now, did you think I was sericus?

With the above information you should be able to find any file anywhere on any disk as long as you have a directory to work from. We will discuss recovery methods in a later chapter, now take a break.



7.0 PASSWORDS AND OTHER TRIVIA

Everybody makes a big deal out of PASSWORDS. I'll admit that I too, at one time, was baffled by the password scheme but within days after getting my copy of "SUPERZAP" all of my disks were without passwords.

If you have read chapter 6.0 you know where the passwords are. First we'll tackle the MASTER DISK PASSWORD.

MASTER DISK PASSWORD

The 'HASH' code for the MASTER DISK PASSWORD is stored in the 'GAT' sector at relative byte 'CD' and 'CF'. In figure 6.2 the 'HASH' for the master disk password is: 'E042'. The MASTER DISK PASSWORD is used by the DOS statement 'PROT :d (LOCK)', where 'd' is a drive specification.

When this command is entered, the MASTER DISK PASSWORD is transferred to ALL user files in the UPDATE and ACCESS PASSWORD bytes. The system files remain as before. (See TRSDOS & DISK BASIC Reference Manual, Section 4, Page 21.)

Conversely '(UNLOCK)' reverses the 'LOCK' process and removes all the passwords that 'LOCK' applied and inserts '9642' into the password bytes.

'9642' is the password, "....." (eight blank spaces), which is ignored by the system - in other words, "....." ('9642') is equivalent to no password at all! The password, "PASSWORD", has the hash code of 'E042'.

We will assume that you have a disk and EVERYTHING is locked out. System files, user files; the whole enchilada. We don't know the master password or we forgot it. At any rate we need access to those files.

- (1) With "SUPERZAP" read a disk with a known password.
- (2) Make a note of the password.
- (3) Remove the known password disk and insert the offending disk.
- (4) Select 'DD' from the "SUPERZAP" menu
- (5) Display track 11, sector 0.
- (6) Using 'MODCE' modify bytes 'CE' and 'CF' to the known password obtained from the 'good' disk.
- (7) Hit break & go to DOS. *(See NOTE, below.)
- (8) Invoke the 'PROT' function.
- (9) Go back to BASIC and 'RUN' "SUPERZAP" and verify that the passwords are changed.
- (10) Take a break, you did good.

* NOTE - If you are using NEWDOS, simply type: CMD"PROT :d (UNLOCK). When the function is completed you will return to BASIC automatically. Then type: CONT <ENTER> Then press: R "SUPERZAP" will continue where it left off without a glitch. VER-R-R-Y fast and handy.

UPDATE & ACCESS PASSWORDS

This is more or less the same routine except that you will modify each password individually. Using the information in chapter 6 locate the proper bytes for the passwords in the 'FPDE' sectors. (Figure 6.8, BYTES 10 & 11 and 12 & 13.) Now, insert '9642' into the UPDATE and ACCESS PASSWORD bytes with the 'MODnn' command in the 'DD' function. You're all done.

Quite a number of people have asked what the algorithm for generating the password is. I don't know and don't care. All I know is that '9642' = " " and is, in effect, no password at all. You may 'remove' all passwords from ALL files including SYSTEM FILES by this method.

OTHER TRIVIA - PROTECT STATUS

If you will remember our discussion in Chapter 6.0 of the directory entries, you will recall that the first byte of each and every 'FPDE' (Figure 6.8, byte 0) contains ALL the 'PROTECT STATUS' information. If you want to remove the 'PROTECT STATUS', change whatever that first byte is, to: 10 (HEX).

If you want to add 'PROTECT STATUS', with "SUPERZAP", then construct a binary number, from the information in chapter 6 figure 6.8, convert it to hex and "ZAP" it into that first byte. There! You're all done again.

MORE TRIVIA - A 'MASTER PASSWORD'

Legend has it that the following 'PASSWORD's will work on any TRSDOS 2.1 'SYSTEM' file:

NV36
F3GUM

I have not tested this but I have it on good authority, that these passwords work.



8.0 DATA RECOVERY PROCEDURES & TECHNIQUES

Your success at data recovery will depend upon your planning ability more than anything else. Whether or not you will successfully recover a file or data will usually depend upon whether or not you have fully thought out just HOW you are going to go about your task, not how well "SUPERZAP" works or whatever utility you decide to use. That brings us to ...

8.1 THE SHELL GAME

Have you ever watched a carnival pitch man work the pea-in-the-shell game? At first it looks simple. There are three shells or dixie cups with the open end down. He places a 'pea' or small white ball under one of the cups. "Now watch closely", he says, and proceeds to switch the cups around in a deliberate manner. "Keep your eye on the shell with the pea", he continues. After half-a-dozen switches, he stops and asks which shell has the pea under it. You have watched him closely and point to one of the shells. He'll ask you if you are sure. You say, "Yes, that's the one!" He picks it up and sure enough, there it is. Now that you feel confident about spotting the pea, you do it again only this time with a little side bet.

Guess what? This time the guy moves the shells so fast you can hardly tell which ones he's moving and when he's finally through with the switches, you have no idea where the pea is. You lose the bet. Convinced that it's really not so tough, you try again and lose again. This will go on until you get smart or run out of money for side bets.

Data recovery is like the shell game. Now you see it, now you don't. If you're watching a real pro, he'll say, "There it is. We'll move it to this track, move up the data 18 bytes, transfer it to here, open it up one sector there, insert this sector here, and copy it back to there." ZAP-BANG! Right before your very eyes, it is fixed. It looks so easy that you decide there's nothing to it. Wrong! Remember, that guy is a pro; you're going to need a little practice before you launch.

The following steps will help you to "keep the 'pea' in sight", so to speak.

1. Determine the cause of the problem.
2. Determine the location of the file on the disk.
Note the location of the FILE EXTENTS.
3. Set up a BUFFER TRACK so you'll have an area to save things to.
4. Look at each sector - determine which sector or sectors are the problem sectors. MAKE NOTES!
5. WRITE DOWN your plan, for recovering the data, in CHECK LIST form.
6. Double check your plan.
7. Format and have standing by, an extra disk so you'll ALWAYS have something to copy to if you find you need extra room.
8. Always work from a BACKUP of the disk or file you are trying to recover.
9. Always check the directory and verify that you are working on the correct disk.
10. NEVER assume anything, (ASS-U-ME; makes an ASS out of U and ME) always CHECK IT OUT FIRST!

11. As you execute each step on your data recovery CHECK LIST, mark it off -- always know where you are and what you are going to do next.
12. Double check your results before copying anything back to its original location.
13. When recovering a data file, make a MAP of the sector to aid in identifying which bytes are what data type.
14. Drink liquids, take aspirin and get plenty of rest.

8.2 USING "SUPERZAP" ON A SINGLE DRIVE SYSTEM.

"SUPERZAP" has its own disk I/O routines and therefore does not need to have a 'SYSTEM DISK' in drive '0'. After "SUPERZAP" loads and executes, you may remove the system disk and put any disk in drive '0'. If you need to transfer sectors from one disk to another, you can do it with the 'SCOPY' command of the 'DD' function.

First read the sector you want to transfer. Then, when the sector is displayed, type 'SCOPY'. When you are prompted to enter the DESTINATION of the 'SCOPY' sector, remove the original disk, from the drive, and substitute your DESTINATION DISK. Finish answering the prompt and the sector will be 'SCOPY'ed to the new disk. It is possible to copy an entire disk this way although it would involve 700 disk swaps for a 35 track disk!

You will find that you only need to copy portions of a file to a new disk, in most cases.

Another technique, is to 'BACKUP' the entire disk and kill everything on the 'BACKUP' disk BUT the file you wish to recover. This will give you plenty of room for 'BUFFER TRACKS'.

8.3 BUILDING A 'BUFFER TRACK'

There's nothing to it. Look on the 'GAT' sector and find an unused track or tracks or GRANULE. Make a note of which tracks or GRANULES are not being used. When you need a place to put something, use those places.

When you are finished using the 'buffer track' you don't even need to remove the material you put there since when the system uses that area it will simply write over it. See, nothing to it.



9.0 FILES - STRUCTURES & TYPES

There are a number of different types of files that may be stored to the disk. Each kind has its own type of 'FORMAT' or 'STRUCTURE'. Being able to recognize a file type, just by looking at the display of the HEX dump, will come with time and a little practice. The following discussion will help you to identify each type of file and understand its structure.

9.1 GENERAL

You cannot tell a file's format by looking in the directory, with one exception: SYSTEM FILES. System files have a special place as well as an 'ATTRIBUTE'. The first two 'FPDE' entry locations, on every directory entry sector, are reserved for SYSTEM FILES. Other than that you will have to know in advance or tell, just by looking, what the file type is.

All file types are written to the disk in 'blocks' of 256 bytes at a time. When there is not enough file material to fill a complete 'block' or sector, the loader finds material from memory (I don't know what the rules are for locating this material) and uses it to pad the sector. For this reason you will not 'see' the end of your files because the last sector will always contain data out to the 'FF' byte.

9.2 ASCII BASIC PROGRAM FILES

We'll start off by looking at our old friend, "SUPERZAP". I have chosen this program because it's one most of you will have and you can experiment on, as I go through each type of file.

An ASCII file, as you can see, appears just as you entered it, as a program, on the display. There are no special loader codes or bytes to speak of. The first byte of an ASCII BASIC program file must be a line number. Each line is terminated with a carriage return (0D HEX). This is how BASIC 'knows' when to start a new line.

At relative byte 'EC', in the below example, is a carriage return. Try "ZAP"ing a '20' (space) into that byte and see what happens when you try to 'RUN' the file. Next try changing the line numbers. The HEXADECIMAL ASCII codes for numbers are:

0 = 30 (HEX)	5 = 35 (HEX)
1 = 31 (HEX)	6 = 36 (HEX)
2 = 32 (HEX)	7 = 37 (HEX)
3 = 33 (HEX)	8 = 38 (HEX)
4 = 34 (HEX)	9 = 39 (HEX)

The 'END OF FILE', of an ASCII file is noted in the directory entry for that file. There is no 'EOF' marker in the actual file. You will also notice that the last sector of the file is full of data down to relative byte 'FF'. This is because 'writes', to the disk, are ALWAYS 256 bytes at a time; NO MATTER WHAT TYPE OF FILE IS BEING WRITTEN.

With very little experimentation you will become familiar with the ASCII BASIC program file.

ASCII coded
line number

ASCII code for
'space' (HEX)

First character of the program text.

'EOR' marker ('0D' HEX)

F00000	3530	2052	454D	3A20	4D41	494E	2F44	4953	50.REM:.MAIN/DIS
F00010	4B20	4D45	4D4F	5259	2044	55AD	502F	4D4F	K.MEMORY.DUMP/MO
F00020	4449	4659	2052	4F55	5449	4F45	2E20	2056	DIFY.ROUTINE...V
F00030	4552	5349	4F4E	2032	2E30	0D31	3030	2047	ERSION.2.0.100.G
F00040	4F54	4F20	3130	3430	3000	3135	3020	4124	OTO.10400.150.A\$
F00050	3D49	4E4B	4559	243A	2049	4620	4124	3D22	=INKEY\$: .IF.A\$="
F00060	2220	5448	454E	2031	3530	3A20	2045	4C53	".THEN.150:..ELS
F00070	4520	4258	3D41	5343	2841	2429	3A20	5245	E.BX=ASC(A\$):.RE
F00080	5455	524E	0D32	3030	2049	4620	4258	203E	TURN.200.IF.BX.>
F00090	3D34	3820	414E	4420	4258	3C3D	3537	2054	=48.AND.BX<=57.T
F000A0	4845	4E20	4258	3D42	582D	3438	3A20	5245	HEN.BX=BX-48:.RE
F000B0	5455	524E	0D32	3530	2049	4620	4258	3E3D	TURN.250.IF.BX>=
F000C0	3635	2041	4E44	2042	583C	3D37	3020	5448	65.AND.BX<=70.TH
F000D0	454E	2042	583D	4258	2D35	353A	2052	4554	EN.BX=BX-55:.RET
F000E0	5552	4E0D	3330	3020	4258	3D2D	0D33	3530	URN.300.BX=-.350
F000F0	204C	5345	3A52	4554	5552	4E3A	5A54	4F50	.LSE:RETURN:STOP

SIMULATE ERROR HERE
(Also see figure 10.3)

||||||| = 'END OF RECORD' (EOR)
===== = LINE NUMBER

9.2 BINARY BASIC PROGRAM FILES

This one is a little tougher to read because line numbers are stored in compressed binary format and the BASIC program statements are in 'TOKEN' form. The BASIC statement 'SAVE' automatically stores your program to the diskette in compressed binary format.

Figure 9.2 is the first sector of "SUPERZAP", as stored in compressed binary format. Compare this sector to figure 9.1. The first thing you will notice is that more program material has been stored in the sector. Compressed binary files are very efficient, in terms of space, and should be used whenever possible.

The first byte of EVERY BASIC FILE in compressed binary format, is 'FF'. If this is any other value you will get the old 'DIRECT STATEMENT IN FILE' displayed on your video, trick. Next I should explain how BASIC 'knows' where the lines are. When the program is in memory the very first part of each line is a 'pointer' to the next

line. This 'pointer' is stored along with the rest of the program material during the 'SAVE' operation. At the end of every line and preceding each pointer for the next line is an 'END OF RECORD' marker. Each BASIC line is a 'record'. The 'EOR' is a hexadecimal '00'.

In the above example I have highlighted the 'EOR', the 'pointer' and the the line numbers. The pointer is not too important. You may make it anything you want but you MUST have something in those bytes. The 'EOR' however, is critical. IT MUST BE '00'.

Try changing the 'EOR' to 'FF' --- the program will 'LOAD' but you'll get gibberish at the end of the line preceding the changed 'EOR' and the next line will be included in the preceding line because there was no way for BASIC to know where the line numbers belonged.

(figure 9.2)

'FF' denotes BASIC
program file.

'68F4' is the 'pointer'
to the next program
line number in RAM.

'3200' is the first line
number of the program - and
is in LSB - MSB order, i.e.,
to be read as '0032' (HEX) =
'50' (DECIMAL).

F00000	FF F4	68 32	00 93	3A20	4D41	494E	2F44	4953	...2...:MAIN/DIS
F00010	4B20	4D45	4D4F	5259	2044	554D	502F	4D4F	K.MEMORY.DUMP/MO
F00020	4449	4659	2052	4F55	5449	4E45	2E20	2056	DIFY.ROUTINE...V
F00030	4552	5349	4F4E	2032	2E30	0000	6964	008D	ERSION.2.0.....
F00040	2031	3034	3030	0029	6996	0041	24D5	C93A	.10400.)...A\$...:
F00050	208F	2041	24D5	2222	20CA	2031	3530	3A20	...A\$."...150:..
F00060	203A	9520	4258	D5F6	2841	2429	3A20	9200	...BX..(A\$):...
F00070	4B69	C800	8F20	4258	20D4	D534	3820	D220	N.....BX...48...
F00080	4258	D6D5	3537	20CA	2042	58D5	4258	CE34	BX..57...BX.BX.4
F00090	383A	2092	0072	69FA	008F	2042	58D4	D536	8:.....BX..6
F000A0	3520	D220	4258	D6D5	3730	20CA	2042	58D5	5...BX..70...BX.
F000B0	4258	CE35	353A	2092	0080	692C	0142	58D5	BX.55:.....,BX.
F000C0	CE42	583A	2092	00AF	695E	0193	3A20	2A2A	.BX:.....:**
F000D0	2A2A	2A2A	2A2A	2A20	5641	5249	4142	4C45	*****.VARIABLE
F000E0	2041	4C4C	4F43	4154	494F	4E20	494E	4849	.ALLOCATION.INHI
F000F0	4249	5445	4400	D569	9001	4432	2528	3129	BITED.....D2%(1)

||||| = 'END OF RECORD' (EOR)
 // = POINTER
 === = LINE NUMBER

Next try changing a 'pointer' to 'FF'. HA! Loaded OK, didn't it! BASIC will take care of this little chore all by itself, even if it's wrong. When the program is 'SAVE'd back to disk, the pointers will be corrected!

Here's the bottom line - when you are "ZAP"ing in a new line number, insert the codes as follows:

'00 FFFF LLMM' - where 'LL' is the LEAST SIGNIFICANT BYTE and 'MM' is the MOST SIGNIFICANT BYTE of the line number you are "ZAP"ing into the sector. "ZAP" the other three bytes as they appear, i.e., '00 FFFF'.

You can experiment by changing these bytes on a copy of "SUPERZAP". Then 'RUN' the changed "SUPERZAP" and by using the 'DM' (DISPLAY MEMORY) function, look at what you have wrought right there in RAM. BASIC will load your program at '6B6C', (HEX) so answer "SUPERZAP"'S prompt, for memory location as '6B00'(HEX).

9.3 'EDITOR ASSEMBLER' SOURCE FILES

To my knowledge, the Apparat 'EDITOR ASSEMBLER' is the only version of the Radio Shack 'EDITOR ASSEMBLER' that writes to a disk file. If there are other versions, they might write the source file differently or use different conventions. In any case, our discussion, here, will concern only the Apparat enhanced 'EDITOR ASSEMBLER'. Figure 9.3 is a typical 'EDITOR ASSEMBLER' SOURCE file.

'EDITOR ASSEMBLER' files are basically ordinary, garden variety, ASCII files. There are some slight differences, however. The first 7 bytes constitute a 'header record'. The first byte (BYTE '0') is always 'D3'. The next 6 bytes are the first six characters of the program name. (I don't know why or what purpose it serves.)

The line numbers are in ASCII format except that 128 (DECIMAL) has been added to the usual ASCII value. For instance an ASCII 'zero' is '30' (HEX) which is equal to 48 (DECIMAL). $48 + 128 = 176$ (DECIMAL) = 'B0' (HEX). In figure 9.3, the line number of the first line of source code is '00100'. You will notice that beginning at relative byte 7, the code reads:

'B0 B0B1 B0B0'. Simply lop off the 'B's and you have '00100'.

It would be a very simple matter to read this file into a BASIC program with the 'INPUT' statement and convert the line numbers back to standard ASCII code, for display, edit the lines and write it back to another file with the 'PRINT #' statement.

The 'EOR' is a carriage return ('0D' HEX), just as it is in a standard ASCII BASIC program file. I suspect that the reason for using the 'B' codes for line numbers is so the file could not be accidentally read into BASIC.

There is an 'EOF' marker at the end of the file. If you note the 'EOF' byte, in the 'FPDE', you will find that the 'EOF' marker, in the file, is one byte less. The 'EOF' marker is '1A' (HEX) and will be preceded by a carriage return.

'D3' denotes that this
is an 'EDITOR ASSEMBLER'
SOURCE file.

First six characters of
the program file name.

Line number - ASCII code
plus 128 (decimal)

First character of
source text.

'EOR' ('0D' HEX)

109500	D345	5045	5242	4FB0	B0B1	B0B0	2009	4F52	.APARBO.....OR
109510	4709	3041	4230	3048	0DB0	B0B1	B0B5	2042	G.0AB00H.....B
109520	4547	494E	0945	5155	0924	0DB0	B0B1	51B0	EGIN.EQU.\$.....
109530	2009	4C44	0948	4C2C	5354	4152	540D	B0B0	..LD.HL,START...
109540	B1B2	B020	094C	4409	2834	3031	3648	292CLD.(4016H),
109550	484C	0DB0	B0B1	B3B0	2009	4C44	0948	4C2C	HL.....LD.HL,
109560	5354	5249	4E47	0DB0	E0B1	B4B0	2009	4C44	STRING.....LD
109570	0928	4255	4646	4552	292C	484C	0DB0	B0B1	.(BUFFER),HL....
109580	B5B0	2009	4A50	0934	3032	4448	0DB0	B0B1JP.402DH....
109590	B6B0	2053	5441	5254	0945	5155	0924	0DB0	...START.EQU.\$..
1095A0	B0B1	B7B0	2009	5055	5348	0948	4C0D	B0B0PUSH.HL...
1095B0	B1B8	B020	094C	4409	484C	2C28	4255	4646LD.HL,(BUFF
1095C0	4552	290D	B0B0	B1B9	B020	094C	4409	412C	ER).....LD.A,
1095D0	2848	4C29	0DB0	B0B2	B0B0	2009	4350	0930	(HL).....CP.0
1095E0	4148	0DB0	B0B2	B1B0	2009	4A50	095A	2C53	AH.....JP.Z,S
1095F0	544F	434B	0DB0	B0B2	B2B0	2009	494E	4309	TOCK.....INC.

9.4 'OBJECT CODE' FILES

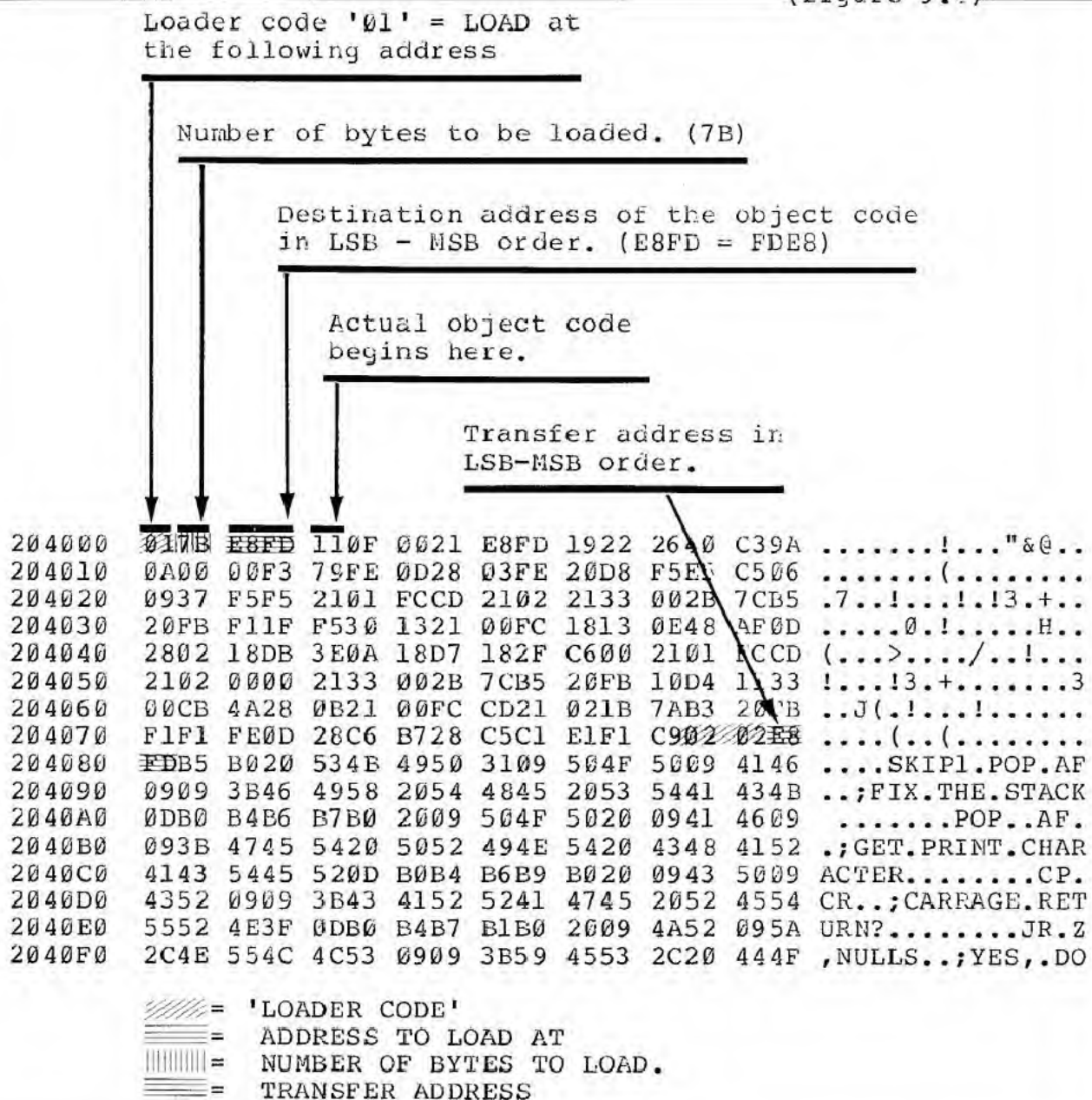
This one is easy to recognize because it never makes sense. The ASCII display portion (the 16 characters on the far right) is seemingly a mish-mash of blanks and random symbols. That's because it's a mish-mash of unprintable characters and random symbols. (WOW! What a lucid explanation!)

I won't attempt to explain the workings of machine language code. I would suggest that you get Bill Barden's latest effort for Radio Shack; "TRS-80 ASSEMBLY-LANGUAGE PROGRAMMING". I know that I have already recommended this book once but what the hell, you didn't rush right out and buy it then, so I thought I'd give it another plug -- it is a good book.

This example is a short (100+ BYTES) machine language printer driver. All machine language object files start with '01' which is also the code for 'load the following bytes for the following number of bytes', at the following address, followed by the actual code.

If you will hark back (you any good at harking?) to the first paragraph of this chapter, you may remember that I pointed out that the machine will only load a maximum of 256 bytes at a time. This is also true of OBJECT files. When the machine has loaded 256 bytes it has to go back, look for another '01' and get the next 256 bytes.

(figure 9.4)



Sometimes, the loader instruction calls for the load to be fewer than 256 bytes, (remember ...256 is represented by '00' (HEX)) in which case the number following the address will be 'FF' or smaller. In the above example, the code is not long enough to call for 256 bytes.

If you look closely, you will see that at relative bytes '7F' and '80' there are the HEXADECIMAL numbers 'E8FD'. This translates to 'FDE8'

and you might recognize these numbers as that first load address that begins at relative byte '02'. In this example, the 'TRANSFER ADDRESS' is the same as the first load address. In another program it could be in another place, depending on where the author of the program decided to begin execution.

There is no 'EOF' marker. The 'EOF' is noted in the 'FPDE' and the last two bytes of the file is ALWAYS the 'TRANSFER ADDRESS'. There are more 'loader codes' and are covered in the next section.

9.5 SYSTEM FILES

System files are just like the 'OBJECT' described in 9.4 with a couple of additions. A system file can be identified from the 'FPDE' by the bit record of the first byte of the 'FPDE'. (BYTE '0', BIT '1' = 1). Other than that there should be no difference between a 'SYSTEM FILE' and any other OBJECT file.

Note, that I said, "Shouldn't be." Well, the Radio Shack system files (originally written by R. COOK) and the VTOS 3.0 system files (written by R. COOK) are different. Figure 9.5 is one of those files. If you poke around, on a NEWDOS disk, you will find that the system files, added by Apparat, are just ordinary object files.

(figure 9.5)

```
000500 0506 5359 5330 2020 1FA9 0D2A 202A 202A ..SYS0.....*.*.*
000510 204E 204F 2054 2049 2043 2045 202A 202A .N.O.T.I.C.E.*.*
000520 202A 0D2A 2050 524F 5052 4945 5441 5259 *.*.PROPRIETARY
000530 2050 524F 4752 414D 202A 0D2A 2043 4F50 .PROGRAM.*.*.COP
000540 5952 4947 4854 2028 6329 2031 3937 3820 YRIGHT.(.).1978.
000550 202A 0D2A 2020 4259 2052 414E 444F 4C50 *.*..BY.RANDOLP
000560 4820 434F 4F4B 2020 202A 0D2A 2020 4341 H.COOK...*.*..CA
000570 5252 4F4C 4C54 4F4E 2C20 5445 5841 5320 RROLLTON,.TEXAS.
000580 202A 0D2A 2041 4C4C 2052 4947 4854 5320 *.*.ALL.RIGHTS.
000590 5245 5345 5256 4544 692A 6A2A 672A 752A RESERVED.*.*.*
0005A0 204E 204F 2054 2049 2043 2045 002A 002A .N.O.T.I.C.E.*.*
0005B0 202A 0D01 080C 40C3 A24B C3B4 4401 0B2D *....@...K..D..-
0005C0 40C3 0044 3EA3 EFC3 BE44 0105 3E40 2101 @...D>....D...>@!..
0005D0 0001 144B 4000 0037 4537 4537 4537 4537 ...K@..7E7E7E7E7E
0005E0 4537 4537 4537 4501 1700 4311 1111 1111 E7E7E7E...C.....
0005F0 1111 1100 0100 0000 0002 0000 00C3 A04B .....K
```

The first thing you will notice about these handsome devils is that they all start with '0506'. Here are the loader codes that I know of -- there could be more.

- 01 = Load the following object code.
- 00 & 03 to 1F = Do not load the following 'n' bytes.
where 'n' = a one byte value of the
number of bytes to skip.
- 0202 = The following 2 bytes contains the transfer
address. See relative bytes '7D' & '7E' in fig-
ure 9.4

So, to translate:

- '0506' = skip the following 6 bytes.
- '1FA9' = skip the following 'A9' bytes.
(169 DECIMAL)

If you will count the 169 (DECIMAL) bytes, you will find that you are at relative byte 'B2'. The very next byte is a code to actually load code. (HOORAY!) It is: '01 080C 40' and is the instruction to load the next 8 bytes at '400C'. Following those 6 bytes we come to another load instruction: '01 0B2D 40', i.e., load the next 11 bytes at '402D'.

Right here you should be saying to yourself, "WAIT JUST A DAMN MINUTE!" RIGHT! The load instruction said to load 8 bytes but there were only 6 bytes before we encountered the next load instruction. In the instruction following that, the load instruction said to load 11 bytes ('0B' HEX) but there were only 9 bytes to the next loader code!

You're right on the ball. The count INCLUDES the two address bytes!

If you 'KILL' a system file, and then decide to re-copy the system file back onto the diskette the 'FPDE' must be in the same position in the directory as on the original DOS! The disk space, assigned to the file, must be accounted for IN THE FIRST EXTENT ELEMENT! No other extent elements may be used! This does not apply to 'SYS6/SYS', it may be anywhere. Also, the actual location of the system program may be anywhere on the disk except those as noted in 9.6, below.

9.6 'BOOT/SYS' - 'DIR/SYS' - SYS0/SYS

'BOOT/SYS' and 'DIR/SYS' are 'SYSTEM FILES' but do not contain code that may be executed. 'BOOT/SYS' is a 'TABLE' that is loaded when LEVEL II BASIC determines that there is a disk drive system attached to the expansion interface. 'DIR/SYS' is the directory and is never executed. They occupy space on the disk, as system files (as they are), but do not contain code that may be executed. 'SYS0/SYS' however, is executable code and must be located beginning at TRACK '0', SECTOR '5'.

9.7 'ELECTRIC PENCIL' FILES.

This one is so easy. It is a straight, plain vanilla, ASCII file, with a carriage return at the end of every record, and an 'EOF' marker at the end of the file. The 'EOF' marker is '00' (HEX) and is located at relative byte '45', in this example. That's all there is to it. Figure 9.6 is a short 'PENCIL' file.

(figure 9.6)

F00000	2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A2A 2A0D	*****.
F00010	2020 2020 2054 4849 5320 4953 2041 4E0DTHIS.IS.AN.
F00020	454C 4543 5452 4943 2050 454E 4349 4C0D	ELECTRIC.PENCIL.
F00030	4649 4C45 0D2A 2A2A 2A2A 2A2A 2A2A 2A2A	FILE,*****
F00040	2A2A 2A2A 0D00 E5E5 E5E5 E5E5 E5E5 E5E5	****.....
F00050	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F00060	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F00070	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F00080	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F00090	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F000A0	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F000B0	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F000C0	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F000D0	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F000E0	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5
F000F0	E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5

9.8 MACRO-80 FILES

Just when I thought I was finished with the file formats I realized that there was one more. Microsoft's MACRO-80 text editor files. This is a line oriented text editor and is not too popular with the TRS-80 crowd but it ain't all that bad either. It has some interesting features (another way of saying, "it's OK, but I prefer to use something else) but is difficult to use and the documentation is not exactly self teaching.

Figure 9.7 is a typical sector of a file created by the MACRO-80 text editor.

(figure 9.7)

```
F00000 B0B0 B1B0 B089 4245 4749 4E20 2020 4551 .....BEGIN...EQ
F00010 5520 2020 2020 3438 3030 300D B0B0 B2B0 U.....48000.....
F00020 B089 2020 2020 2020 2020 4F52 4720 2020 ..... .ORG...
F00030 2020 4245 4749 4E0D B0B0 B3B0 B089 2020 ..BEGIN.....
F00040 2020 2020 2020 4C44 2020 2020 2020 484C .....LD.....HL
F00050 2C30 4139 4448 0DB0 B0B4 B0B0 8920 2020 ,0A9DH.....
F00060 2020 2020 204C 4420 2020 202 2028 4646 .....LD.....(FF
F00070 4646 292C 484C 0DB0 B0B5 B0B0 8920 2020 FF),HL.....
F00080 2020 2020 2045 4E44 2020 2020 2042 4547 .....END.....BEG
F00090 494E 0D00 0000 0000 0000 0000 0000 0000 IN.....
F000A0 0000 0000 0000 0000 0000 0000 0000 0000 .....
F000B0 0000 0000 0000 0000 0000 0000 0000 0000 .....
F000C0 0000 0000 0000 0000 0000 0000 0000 0000 .....
F000D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
F000E0 0000 0000 0000 0000 0000 0000 0000 0000 .....
F000F0 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

There are only a few minor differences between this file structure and ASCII files, Apparat modified EDITOR/ASSEMBLER files and 'ELECTRIC PENCIL' files. Like ASCII files and 'ELECTRIC PENCIL' files, each record is terminated with a carriage return ('0D' HEX). Like an 'ELECTRIC PENCIL' file, the end of the file contains a zero. This text editor, unlike the Apparat version of 'EDITOR ASSEMBLER', DOES NOT WRITE 'FILLER MATERIAL' TO THE SECTOR - IT WRITES ZEROS FOR FILLER.

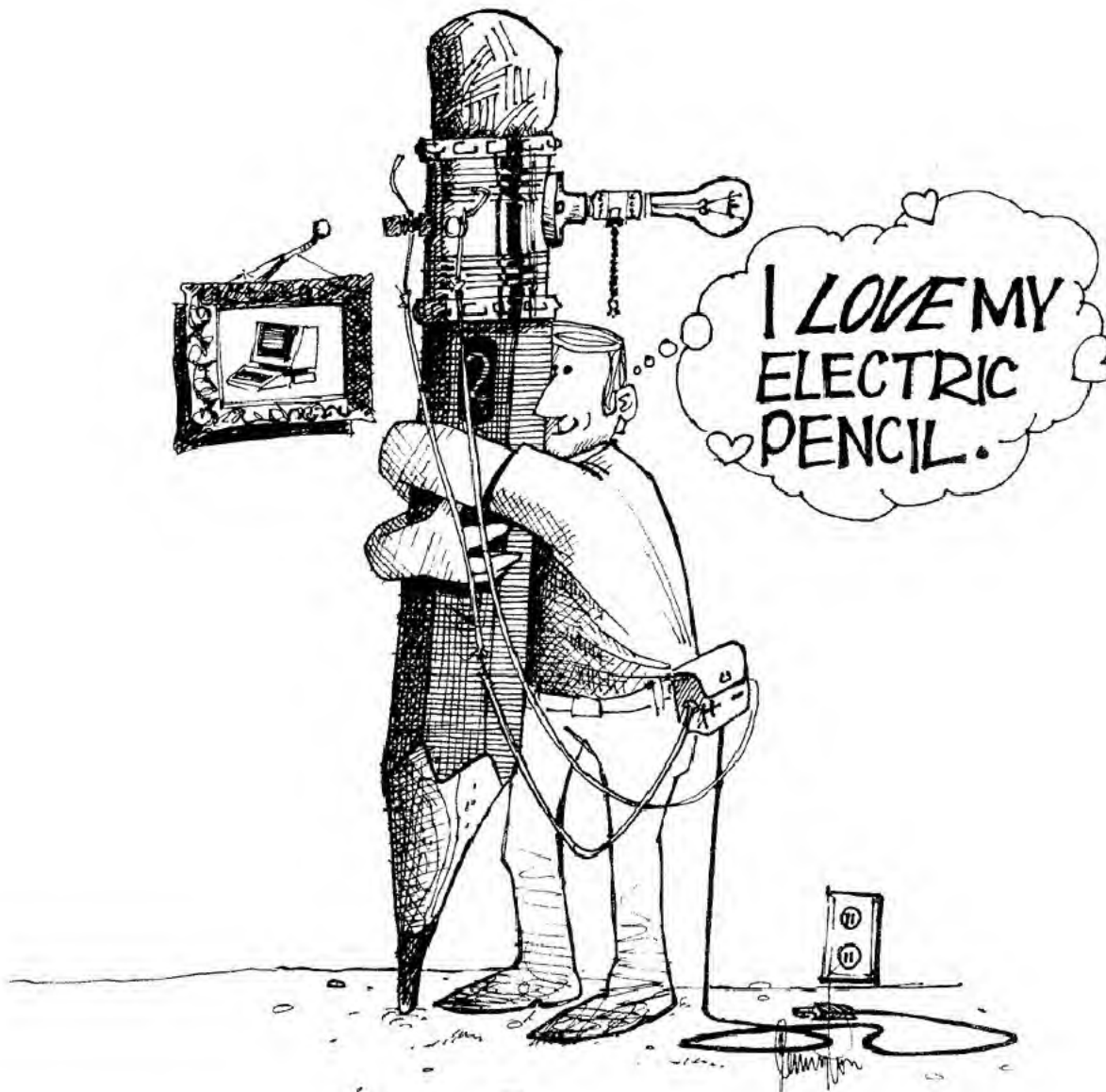
This makes the MACRO-80 files compatible with 'ELECTRIC PENCIL'. After a MACRO-80 file is loaded into 'PENCIL', it must be modified somewhat in order to print on your line printer. After loading a MACRO-80 file into 'ELECTRIC PENCIL', you will notice that the MACRO-80 line numbers appear on the screen as TRS-80 graphics.

Now here is some real wizard stuff - with the 'DOWN ARROW', move the cursor past the text. As the cursor passes each group of graphics, the graphics are changed to numbers! How about that!

Following each number there will be a small 'RIGHT ARROW'. (ASCII code number: '89' HEX) This arrow is NOT one of the normal 'ELECTRIC PENCIL' codes and must be removed to print the file on the line printer while in 'ELECTRIC PENCIL'.

Once the 'right arrow' is removed, however, you cannot reload the file into the MACRO-80 text editor without MACRO-80 putting on another set of line numbers. You are better off to remove all line numbers in the pencil file and let MACRO-80 re-append a new set.

You will notice that there is no 'HEADER RECORD' on MACRO-80 files, as there are on Apparat generated 'EDITOR ASSEMBLER' files. MACRO-80 will load any kind of an ASCII file, and will attach its own line numbers during the load process. The file may be written back to disk from MACRO-80, using the 'SWITCH' option, which will delete any line numbers it has attached. This means that you could make 'ELECTRIC PENCIL' files from MACRO-80 directly, if you had the need.



10.0 DATA RECOVERY

If you are like most people, you are reading this first instead of last. If you are, I can only say, "Good Luck."

You really need to get a good understanding of the disk and the directory before you try these things. Now, no matter how painful, go back and read the first eight chapters.

Now that you've read the first eight chapters (Jeez, that was fast!) we will proceed ..

10.1 RECOVERING A 'HASH' CODE FOR THE 'HIT' SECTOR

Since we do not know the algorithm for the 'HASH CODE' we will have to revert to devious means to obtain it. There are two ways:

- (1) 'SAVE' a one line program, while in BASIC using the 'FILENAME' of the program whose 'HASH' you wish to obtain ONTO A SEPARATE DISK. Example:
10 REM THIS IS A TEST
SAVE"<FILENAME/EXT>"

The reason for a separate disk is so that the DOS won't accidentally write over the file name 'FPDE' you are trying to recover! Of course, if you have copied the sector to a 'BUFFER TRACK' there is no problem.

- (2) 'OPEN' a file in command mode, from BASIC, using the 'FILENAME' of the program whose 'HASH' you wish of obtain. Example:
OPEN"O",1,"<FILENAME/EXT>" <ENTER>

Once you have 'created the file', use "SUPERZAP" to go look at the 'HASH' (don't forget to write it down), then 'KILL' the file. Now you have the 'HASH CODE' for the file you wish to recover.

10.2 RECOVERING A 'KILLED FILE'

When you 'KILL' a file the following three things happen:

- (1) The 'HASH' code is removed from the 'HIT' sector.
- (2) The 'GAT TABLE' is revised to reflect the now available granules.
- (3) Byte 0 in the 'FPDE' (and 'FXDE', if it exists) is changed to '00'

Everything else remains as it was. The file is still out on the disk, and the entries in the 'FPDE/FXDE' remain unchanged except for byte 0.

```
***** WARNING *** WARNING *** WARNING *****
**
**      ON TRSDOS 2.2 ALL TRACES OF THE DIREC-      **
**      TORY ENTRIES IN THE 'FPDE/FXDE' ARE          **
**      ZEROED OUT WHEN A FILE IS 'KILL'ED !!        **
**
***** WARNING *** WARNING *** WARNING *****
```


As you can observe in my casual note above, Radio Shack's Software Development (?) Group, has, once again, in its infinite wisdom (I wonder if they talk to the Ayatollah?), has done a really neat thing for the users. I suppose this is to protect them from having one of their "super-neat" programs, like the 7 hour / 400 name / MAILLIST / sort, slipped out, recovered, and unscrupulously used by a demented inmate of the Peoria Institute for the Rehabilitation of the Non-Mentally Deranged and former residents of Burbank.

You can still recover 'KILL'ed files on TRSDOS 2.2 but you will have to search around until you find all of the sectors, then reconstruct an 'FPDE' in the directory.

10.2.1 Here are the steps for
recovering a 'KILL'ed file:

- (1) Obtain the 'HASH' code of the 'FILENAME' of the program. See 10.1 above.
- (2) "ZAP" the hash code into the proper place in the 'HIT' sector.
- (3) "ZAP" byte 0 of the 'FPDE/FXDE' with a '10'

10.2.2 If it is a 'BASIC PROGRAM'
(BINARY or ASCII format) file
then:

- (1) 'LOAD' the file into BASIC
- (2) 'SAVE' the file using the 'FILENAME' it was recovered under. This will correct the 'GAT' allocation.
- (3) Run a 'DIRCHECK' to determine if any 'GAT' or 'HIT' errors remain.

```
***** NOTE *****
**                                                     **
**   FILES WILL LOAD AND EXECUTE PROPERLY           **
**   WITH 'GAT' ERRORS -- BEWARE OF 'SAVE'          **
**   OPERATIONS WHEN 'GAT' ERRORS EXIST!            **
**                                                     **
*****
```

10.2.3 If it is an 'ASCII', 'BINARY'
(RANDOM) data or an ASSEMBLY
LANGUAGE program or data file
then:

- (1) Run 'DIRCHECK' and obtain a listing of the 'GAT' and/or 'HIT' errors that exist.
- (2) Using the 'GAT MAP' (figure 6.6) correct 'GAT' errors by "ZAP"ing the 'GAT' table.
- (3) Repeat (1) until all 'GAT'/'HIT' errors are corrected.

10.3 RECOVERING A FILE/DISK THAT WON'T 'BOOT' OR READ THE DIRECTORY.

This one can be a bitch, to say the least. There are no short cuts save one, and that one dictates that you have a 'BACKUP' copy of the disk with a directory that is partially correct. This will give some clues as to the track locations of the various file 'EXTENTS'. Other than that, it's time for that wonderful programmers' pastime, "SEARCH THE DISK"!

Be sure to take plenty of notes if you have to search the disk sector by sector; if you don't, you won't remember which sectors you searched and tried and which ones you didn't.

I see that Bill Barden has a question. Yes, Bill? How does the directory come to be 'EATEN' in the first place? Hmmm, good question. There are dozens of reasons and the principal ones are:

- (1) You tried to 'KILL' an open file. (DIRECTORY CLOBBERED)
- (2) You turned on the disk drive with the disk in the drive. (USUALLY TRACK 0, SECTOR 0 BUT COULD BE THE DIRECTORY)
- (3) You turned your CPU/INTERFACE on, with the disk in the drive. (WHEREVER THE DISK HEAD WAS LOCATED AT THE TIME)
- (4) You attempted to 'SAVE' or write to the last sector of a nearly full diskette (TRSDOS 2.1 & 2.2) - (It should work but doesn't) (DIRECTORY GONE AND USUALLY CONTAINS 'GARBAGE')
- (5) DOS got confused during a 'CLOSE' operation and de-allocated a few GRANULES. After several 'SAVES' and 'CLOSE's it became further confused; didn't quite know where to put something for a 'SAVE', 'PUT' or 'CLOSE' so it deposited some of its burden all over the directory making it unreadable. (DIRECTORY USUALLY CONTAINS PROGRAM MATERIAL)
- (6) Your CPU or Expansion Interface has bad memory and/or the file control block has bad data prior to or during a 'CLOSE'. (DIRECTORY CONTAINS GARBAGE)
- (7) Faulty logic card in the disk drive unit. (DIRECTORY CONTAINS GARBAGE)
- (8) Disk head out of alignment on a drive unit. (DIRECTORY HAS PARITY ERRORS, SECTOR NOT FOUND ERRORS)
- (9) Someone in Fort Worth dosen't like you. (NOTHING IS RIGHT)
- (10) Everyone likes you, your system works perfectly but you don't want to be left out of all this fun.

10.3.1 The following steps are for
recovering a totally 'eaten'
or 'clobbered' directory.

You can usually spot this one right away because the directory contains 'garbage' or program material. You will find that you will have no trouble reading the directory with "SUPERZAP", but the disk won't function at all.

- (1) Locate the file EXTENTS; where they start; where they end; and the exact number of sectors in the file.
- (2) 'FORMAT' a 'working disk'
- (3) Create a file FPDE on the working disk, with the 'SAVE' or 'OPEN' technique. (See 10.0 above)
- (4) Transfer the sectors that you previously located from the clobbered disk to the working disk INTO ONE HUGE FILE EXTENT IF POSSIBLE. This will make fixing the directory easier because you need make only one EXTENT in the 'FPDE'.
- (5) "ZAP" the EXTENT of the 'FPDE' to point to the reconstructed file.
- (6) "ZAP" the EOF SECTOR BYTES (BYTES 14 & 15) with the sector count +1.
- (7) "ZAP" the EOF BYTE (BYTE 3) with the relative byte number of the last byte in the EOF sector. If you cannot identify the EOF byte, then make a 'SWAG' (Scientific Wild-Ass Guess). If it's wrong you can always change it.
- (8) If its a 'BASIC PROGRAM' file, 'LOAD' it and see how much of it loads. You should be able to tell how successful you were just by looking at a 'LIST' of the program. If some of it is garbage note the last 'good' line of the program and RUN "SUPERZAP" again and take another crack at eliminating the bad portions by moving up or relocating other good sectors to substitute into the bad sectors.
- (9) If some portions of a sector are 'bad' then use 'COPY DISK DATA' to move good data over bad.
- (10) If the recovered material is other than a 'BASIC PROGRAM' file, then you will have to verify data by reading it in via 'LINEINPUT' or 'FIELD' statements.
- (11) You may attempt to verify ASSEMBLER LANGUAGE files by using APPARAT'S DISASSEMBLER. If they make sense to you in the disassembled state (Which requires an intimate knowledge of assembler code) then the file is probably OK. If not, you will have to execute the file and attempt to locate problems via 'DEBUG'.
(I wonder if Muhammad Ali knows about DEBUG?)

10.3.2 Recovering a file/disk with an unreadable directory.

This one is fairly easy but I should describe one of the fine points that exists here. You may find that the disk won't 'BOOT' (if it's a system disk) or that you can't get a 'DIR' to work but you can still read all the sectors with "SUPERZAP", then you have an 'UNREADABLE' directory. It is 'UNREADABLE' by the system; NOT by "SUPERZAP"! Here are just a few of reasons why a directory is 'unreadable'.

- (1) It isn't the directory at all - the 'BOOT' sector is 'clobbered'.
- (2) One or more of the directory sectors has a PARITY error.
- (3) READ protect status has somehow been removed from one or more of the directory sectors.

Before going into outer space to fix the problem first try using "SUPERZAP"'S 'BACKUP' function. Many times this will 'fix' the disk without further adieu, If that shot in the dark fails, try the following:

To repair the problem you must correctly identify it. Using "SUPERZAP"'S 'DD' function:

- (1) Check the directory sectors for parity errors. They will automatically be detected when you try to read a sector with bad parity. If, after a bad parity read, the sector looks OK, 'SCOPY' it back to THE SAME SECTOR YOU READ IT FROM. This will automatically repair the bad parity!
- (2) Check the sectors for 'READ' protect status. There will be a '6' on the last line of the "SUPERZAP" display on the left side in column 7. IF THIS '6' IS NOT THERE 'READ' PROTECT STATUS HAS BEEN REMOVED AND MUST BE REPLACED!
 - (a) Copy the sector to a 'BUFFER SECTOR'.
 - (b) Copy ANY GOOD DIRECTORY SECTOR TO THE SECTOR WITH BAD STATUS. This will re-establish 'READ' protect status.
 - (c) Copy the sector in the 'BUFFER SECTOR' back to its original sector using the 'COPY DISK DATA' function. *** NOTE: Only copy 255 bytes ('FE' (HEX)) back to the original sector! This will preserve the 'READ' protect status.
 - (e) Manually "ZAP" the 256th byte back into place.
- (3) If after checking the directory, all is in order, and you determine that it MUST be the 'BOOT' sector (Track 0, Sector 0); simply copy a good track 0, sector 0 from another disk.

10.4 RECOVERING AN ELECTRICALLY DAMAGED DISK

The disk is OK, you can detect no physical problems, but some tracks or sectors will not load. It might have become that way by getting 'zapped' with static, or you performed a 'WRITE' operation on a disk that was not centered in the drive. Also most of the reasons given for a 'clobbered' directory, in 10.3, could apply here. In addition, there are some other less obvious reasons which I shall call to your attention.

- (1) Beware of magnetic paperclip holders! These things are common items around offices and they will make a meal out of your disks.
- (2) Magnetized paper clips that have been in magnetic paperclip holders. Don't ever let anyone use paperclips on disks! Besides, it's bad for diskettes even if the paper clips are not magnetized.
- (3) A disk placed under a telephone is a likely candidate for the format farm. It can be wiped out when the phone rings.
- (4) A disk placed next to an electric pencil sharpener or any other type of device with an electric motor or transformer can be erased.

An uncentered disk is a common problem. Shugart, Pertec, and Wangco disk drives have this problem. It is due to the short centering cone, and abrupt lead-taper of the centering cone. It can be partially alleviated by NOT CLOSING THE DOOR TO THE DISK DRIVE UNTIL THE MOTOR GOES ON! However, for you TRSDOS 2.1 and 2.2 users this will result in the famous old 'SILENT DEATH' routine if you don't get the door closed before the DOS accesses the designated drive. APPARAT NEW DOS and VTOS 3.0 will wait until you get the door closed. VTOS 3.0 is a little more fussy but <SHIFT> 'BREAK' will cause it to try again.

Now for the recovery. Usually you need to recover the disk because you need to back it up and a sector is bad in the middle of a particular program or file. (What else?)

- (1) Use "SUPERZAP" to verify the sectors.
- (2) Note the bad sectors.
- (3) Format a disk and make a "SUPERZAP" BACKUP.
- (4) 'SKIP' any bad sectors that won't respond to a 'REENTRY'.
- (5) Using 'DD' look at the sector BEFORE and AFTER the bad sectors -- maybe they don't contain anything important anyway, in which case you can forget them.
- (6) If you're not sure that the bad sectors are being used by some file, check the 'GAT' sector and determine if the track/sector is allocated. If you want to find out which file it's allocated to, without plowing through the directory, simply de-allocate the track with "SUPERZAP" and run 'DIRCHECK' --- It will tell you which file it was allocated to.
- (7) Attempt to LOAD the program or file that was in the damaged portion of the disk.
- (8) Follow steps outlined in 10.3.1 (8), (9), and (10)

10.5 RECOVERING A PHYSICALLY DAMAGED DISK

This unfortunate circumstance can occur in many unpredictable ways. You could have accidentally scratched the disk while using it for a shoe horn or a pipe cleaner. It could have been carelessly handled by a store clerk who thought it was a Master Charge Card and ran it through his little machine. A friend who had just dropped by to visit after a taco eating contest picked it up with thumb and forefinger expertly placed on the head access slot.

In any case, this aggravation is handled in exactly the same way as described in 10.4.

CAUTION - MAKE SURE THE FOREIGN MATERIAL THAT IS ON THE DISK IS SUFFICIENTLY ATTACHED SO AS NOT TO CONTAMINATE THE DISK 'READ/WRITE' HEAD. If you determine that the disk surface has been contaminated with a foreign substance such as finger prints, coffee, hand lotion etc., here is one semi-drastic measure you may take that I have used successfully on one or two occasions:

- (1) CAREFULLY slit the back of the disk jacket and remove the disk. DO NOT TOUCH THE DISK SURFACE! HANDLE BY EDGES AND CENTER ONLY!
- (2) CAREFULLY wash the disk in warm soapy water using your WET AND SOAPY fingers to GENTLY STROKE (DO NOT RUB) the disk.
- (3) THOROUGHLY rinse the disk in warm water.
- (4) If soap and water did not do the job, add alcohol to the water and try again.
- (5) Repeat # 3.
- (6) Place the disk on a sheet of NEWSPAPER. WARNING - PAPER TOWELS LEAVE LINT! Lay another sheet on top. Press gently. Repeat until the disk is dry.
- (7) Under no circumstances rub the disk!
- (8) When the disk is dry, CAREFULLY reinsert it into A NEW JACKET - DO NOT TOUCH THE MEDIA! (Here is a good use for those diskettes that weren't any good and you couldn't bring yourself to throw away.)
- (9) BACKUP THE DISK IMMEDIATELY!

10.6 RECOVERING A 'BAD PARITY' ERROR

Bad parity can be the result of one bit being incorrect or as bad as every bit EXCEPT one. Sometimes the sector is good and the parity is incorrect!

If you notice that one drive has more parity errors than another then look to the drive as the cause. You may also find that the sectors are OK and that you only get errors during 'READ' operations. Once again, look to the drive unit for the fault.

- (1) Using the "SUPERZAP" 'VERIFY DISK SECTORS', determine which sectors are bad. If there are only a few of them then
- (2) Use the 'DD' function to read the sector. If everything looks OK and you cannot detect an error, then type: 'MOD00' then press <ENTER>. This will simply write the sector back to the disk WITH CORRECT PARITY. DO NOT ACTUALLY 'MODIFY' ANYTHING!

If the problem cannot be corrected by the above method then:

- (1) Attempt a "SUPERZAP" 'BACKUP'. Use the 'R' (RE-ENTRY) command when the routine encounters a sector with 'BAD PARITY'. If you are unable to copy the sector then make a note of the unreadable sector(s) and 'SKIP' those unreadable sectors.
- (2) Determine if the 'BAD PARITY' sectors are actually used by a file. There is no use in recovering a sector not used by anything. Once you have made a "SUPERZAP" 'BACKUP', onto a formatted diskette, all the sectors are good and the disk will 'BACKUP' via normal methods.
- (3) If the 'BAD PARITY' sector(s) are used by a file then there are two procedures we can use to recover the file.

METHOD 1

- (a) Attempt to read the sector with the 'DD' function. If the read is fairly successful, 'SCOPY' the displayed sector to a 'BUFFER TRACK' or 'BUFFER SECTOR'.
- (b) Continue to attempt reads with 'DD' and copy partially read sectors to the 'BUFFER' with 'SCOPY' until you are satisfied that you cannot get any more good bytes from the sector.
- (c) Using the 'PD' function, make a hard copy of the 'BUFFER' sectors. With this as a guide ...
- (d) Painfully construct the sector byte-by-byte, using the 'MODm' function, to yet another 'empty' 'BUFFER' sector. Or, if the 'SCOPY'ed sectors have large chunks of usable material, then use one of these sectors for reconstruction. You may also use 'COPY DISK DATA' for moving bytes from one sector to another.
- (e) When the sector is reconstructed, copy it back to its original track/sector address.

METHOD 2

- (a) Find an earlier version of the clobbered sector and copy those bytes to the bad sector.
- (b) If the earlier version is incomplete and you simply need to recover MOST of the file, then move the sectors below the offending sector up and change the 'FPDE' pointer in the directory to reflect the current Sector count and the EOF BYTE.

10.7 RECOVERING A 'DIRECT STATEMENT IN FILE' ERROR

I must confess that the first hundred times I encountered this error it nearly drove me crazy. (Nearly?) Before I got the TRS-80, I had never laid hands on a computer in my life and the cryptic messages from this magic machine, without explanations, were completely baffling.

What made it doubly worse is the fact that neither the LEVEL II manual or the disk manual gave the slightest clue as to what a 'DIRECT STATEMENT IN FILE' was or how it got there.

This little cutie may occur in one of two ways. It is usually the result of a very minor 'bug' in LEVEL 2 BASIC. It happens when you 'SAVE' a program that has a statement line that is longer than 240 bytes.

How can that happen? Easy. It happens when you 'EDIT' a long line and insert more characters than the disk operating system can handle. Normally the system checks line lengths and will not allow you to make a line too long. In the 'EDIT' mode however, the checking does not function quite correctly.

The other condition is very similar to the 'EDIT' condition, in that you 'SAVED' a file WITH THE ASCII OPTION, and it had statement lines that were longer than 240 bytes WHEN THE BASIC TOKENS WERE EXPANDED TO THEIR FULL ENGLISH EQUIVALENT!

In the TRS-80 LEVEL II manual, Appendix A, page 16 it clearly states:

Program Line Length: Up to 255 characters.

Actually BASIC will only 'LOAD' 240 characters of program material at a time! An assembly language 'OBJECT CODE MODULE' will load 256 characters of program material. A random file record or an ASCII data record, on the other hand, will load up to 255 characters with TRSDOS 2.1 and NEWDOS 2.1 and 256 characters with TRSDOS 2.2. SUPERDOS will load up to 4,095 characters with certain types of files and 256 characters with random files.

For a BASIC program, each statement line must have a line number. The condition that exists with a 'DIRECT STATEMENT IN FILE' is that the computer loaded a line with a line number and 240 characters and there were some characters left over. These are the 'DIRECT STATEMENTS' that are in the file. Since they don't have line numbers, BASIC doesn't know what to do with them!

What does that have to do with the ASCII mode? Well, Level 2 BASIC actually uses 'TOKENS' to store program statements in memory. For instance, when you type 'PRINT' the machine does not store the actual characters that you typed or that it is displaying on the video. It is actually storing a '?' in memory. This '?' takes only one byte to store, the word 'PRINT' would take 5 bytes to store. (See Appendix A for a complete listing of the LEVEL II 'BASIC TOKENS')

When you are writing a program, the system keeps track of how many characters each 'TOKEN' would take if it were completely spelled out. This would NORMALLY prevent you from getting a direct statement in file when you 'SAVE' a program file with the ASCII option. In the 'EDIT' mode, LEVEL 2 will allow you to insert a few extra characters --- just enough to put you over the legal limit. There you have it, friends and neighbors --- the Secret of the Shifting Whispering Sands.

Now, what to do about it. Actually this is a fairly easy condition to fix. All we need to do is insert a line number in front of the offending 'DIRECT STATEMENT' that's in the file. We'll do the easy one first.

10.7.1 ASCII file with direct statement error.

- (1) Determine the last line number that 'LOAD'ed.
- (2) Determine the last characters that 'LOAD'ed.
(Use 'LIST' to determine (1) and (2).)
- (3) Locate the file on the disk, using the previously described methods in 10.0.
- (4) Scan the sectors of the file until the sector with the error is found. This will be easy with an ASCII file because everything, including line numbers, are in readable form.
- (5) Now, "ZAP" a line number anywhere in the offending line that is LARGER than the preceding line number and SMALLER than the next line number.
You will lose a few characters of your program.
(A small price to pay.)

10.7.2 A 'BINARY' file with 'DIRECT STATEMENT' error.

This one isn't really so tough - it's just that the file display is a little harder to read. All of the line numbers are in hexadecimal notation and the statements are in token form. You should be able to recognize portions of the program however, from the variable statements, string statements and remarks.

A 'BINARY' file, with a direct statement error is a very rare occurrence. It has happened to me only a few times in a year. I do not know how I was able to generate the error and I have not been able to duplicate the error on purpose but have had it happen accidentally several times. Because I was not able to duplicate the error on purpose, the following examples are contrived, but the recovery is a valid one ... I know THAT for a fact as a result of having had to recover a couple of binary files with a direct statement in file error.

Figure 10.1 is an example of a BASIC program file stored in 'BINARY' format. You will recognize the code as the first part of your "SUPERZAP 2.0" program so you will be able to experiment along with me, as we try out these various techniques. Figure 10.3 is a listing of the first part of "SUPERZAP", with and without the simulated 'errors', so you may compare the actual 'BASIC CODE' with what is stored on the diskette.

(figure 10.1)

```
01A000 FFF4 6832 0093 3A20 4D41 494E 2F44 4953 ...2...:MAIN/DIS
01A010 4B20 4D45 4D4F 5259 2044 554D 502F 4D4F K.MEMORY.DUMP/MO
01A020 4449 4659 2052 4F55 5449 4E45 2E20 2056 DIFY.ROUTINE...V
01A030 4552 5349 4F4E 2032 2E30 0000 6964 008D ERSION.2.0.....
01A040 2031 3034 3030 0029 6996 0041 24D5 C93A .10400.)...A$...
01A050 208F 2041 24D5 2222 20CA 2031 3530 3A20 ...A$."...150:..
01A060 203A 9520 4258 D5F6 2841 2429 3A20 9200 ....BX..(A$):...
01A070 4E69 C800 8F20 4258 20D4 D534 3820 D220 N.....BX...48...
01A080 4258 D6D5 3537 20CA 2042 58D5 4258 CE34 BX..57...BX.BX.4
01A090 383A 2092 0072 69FA 008F 2042 58D4 D536 8:.....BX..6
01A0A0 3520 D220 4258 D6D5 3730 20CA 2042 58D5 5...BX..70...BX.
01A0B0 4258 CE35 353A 2092 0080 692C 0142 58D5 BX.55:.....,BX.
01A0C0 CE42 583A 2092 953A 923A 9493 3A20 2A2A .BX:.....:..**
01A0D0 2A2A 2A2A 2A2A 2A20 5641 5249 4142 4C45 *****.VARIABLE
01A0E0 2041 4C4C 4F43 4154 494F 4E20 494E 4849 .ALLOCATION.INHI
01A0F0 4249 5445 4400 D569 9001 4432 2528 3129 BITED.....D2%(1)
```

In the above figure, relative byte '3D' and '3F' are typical line numbers. The contents of these two bytes are '64' and '00'. To read them you must REVERSE THEIR ORDER so that they read as '00' and '64'. If you have done your homework and didn't chew gum in class, you know that 0064 (HEX) is equal to 100 (DECIMAL).

Our simulated 'DIRECT STATEMENT' error is the code beginning at relative byte 'C6' and continues for the next 5 bytes. Actually I have 'rigged' this error but you may verify that the changes are valid by "ZAP"ing the error onto a backup copy of "SUPERZAP" and 'LOAD'ing it --- THIS WILL LOAD; AN ACTUAL 'DIRECT STATEMENT IN FILE' ERROR WILL LOAD ONLY UP TO THE POINT WHERE THE ERROR EXISTS! Now, "ZAP" in the correction and 'LOAD' it again.

The exact error is '953A 923A 94'. In actual practice you will not know the exact error or precisely where it occurs. All that you will know is that the program won't 'LOAD' beyond a particular PLACE in a line number. That is your clue as to where to 'fix' the damn thing.

Since the exact place that we want to "ZAP" in a new line number is relatively unimportant, I'll pick relative byte 'C1' and start making the changes there. We need a line number larger than 300 and less than the next line number (which happens to be 400); I think 350 is a good choice. Note that 350 (DECIMAL) is equal to 015E (HEX). In keeping with the general scheme of things we must reverse the order of the HEX numbers so they read: '5E01'. In addition, we need to insert the codes that BASIC needs to properly load each line.

The codes are in the 3 bytes preceding every line number and always start with '00'. Since we need to just get the file loaded so we can correct it, simply 'borrow' a code from another line number ('0080 69' is the code from line 300) and you have everything you need to complete the operation.

SHAZAM! We start "ZAP"ing relative byte 'C1' with the following: '00 8069 5E01'. Figure 10.2 is how the sector will look after the new line number is inserted.

(figure 10.2)

```

01A000 FFF4 6832 0093 3A20 4D41 494E 2F44 4953 ...2...:MAIN/DIS
01A010 4B20 4D45 4D4F 5259 2044 554D 502F 4D4F K.MEMORY.DUMP/MO
01A020 4449 4659 2052 4F55 5449 4E45 2E20 2056 DIFY.ROUTINE...V
01A030 4552 5349 4F4E 2032 2E30 0000 6964 008D ERSION.2.0.....
01A040 2031 3034 3030 0029 6996 0041 24D5 C93A .10400.)...A$...
01A050 208F 2041 24D5 2222 20CA 2031 3530 3A20 ...A$."...150:.
01A060 203A 9520 4258 D5F6 2841 2429 3A20 9200 ...BX..(A$):...
01A070 4E69 C800 8F20 4258 20D4 D534 3820 D220 N....BX...48...
01A080 4258 D6D5 3537 20CA 2042 58D5 4258 CE34 BX..57...BX.BX.4
01A090 383A 2092 0072 69FA 008F 2042 58D4 D536 8:.....BX..6
01A0A0 3520 D220 4258 D6D5 3730 20CA 2042 58D5 5...BX..70...BX.
01A0B0 4258 CE35 353A 2092 0080 692C 0142 58D5 BX.55:.....,BX.
01A0C0 CE00 8069 5E01 953A 923A 9493 3A20 2A2A .BX:.....:..**
01A0D0 2A2A 2A2A 2A2A 2A20 5641 5249 4142 4C45 *****.VARIABLE
01A0E0 2041 4C4C 4F43 4154 494F 4E20 494E 4849 .ALLOCATION.INHI
01A0F0 4249 5445 4400 D569 9001 4432 2528 3129 BITED.....D2%(1)

```

Load the file, correct the line we just created, and the line preceding it. Now 'SAVE' it back to the disk and everything will be correct. With that complete you are ready to run. (Next case!)

(figure 10.3)

NORMAL "SUPERZAP" LISTING

```
50 REM: MAIN/DISK MEMORY DUMP/MODIFY ROUTINE. VERSION 2.0
100 GOTO 10400
150 A$=INKEY$: IF A$="" THEN 150: ELSE BX=ASC(A$): RETURN
200 IF BX >=48 AND BX<=57 THEN BX=BX-48: RETURN
250 IF BX>=65 AND BX<=70 THEN BX=BX-55: RETURN
300 BX=-BX: RETURN
350 REM: ***** VARIABLE ALLOCATION INHIBITED
400 D2%(1)=VARPTR(D2%(5)): DEFUSR2 = VARPTR(D2%(0))
450 X=USR2(0): RETURN
500 ' ***** END OF VARIABLE ALLOCATION INHIBIT
550 GOSUB 150: GOTO 200
```

"SUPERZAP" LISTING WITH SIMULATED ERROR CORRECTED

Notice that line 350 now contains 'GARBAGE' but file will load OK.

```
50 REM: MAIN/DISK MEMORY DUMP/MODIFY ROUTINE. VERSION 2.0
100 GOTO 10400
150 A$=INKEY$: IF A$="" THEN 150: ELSE BX=ASC(A$): RETURN
200 IF BX >=48 AND BX<=57 THEN BX=BX-48: RETURN
250 IF BX>=65 AND BX<=70 THEN BX=BX-55: RETURN
300 BX=-
350 LSE:RETURN:STOPREM: ***** VARIABLE ALLOCATION INHIBIT
400 D2%(1)=VARPTR(D2%(5)): DEFUSR2 = VARPTR(D2%(0))
450 X=USR2(0): RETURN
500 ' ***** END OF VARIABLE ALLOCATION INHIBIT
550 GOSUB 150: GOTO 200
```

10.8 RECOVERING DATA FILES

There are no special things to know about data files that make them more or less difficult to recover than any other type of file. There are two formats for data: (1) ASCII, (2) compressed binary. The 'FPDE' and 'FXDE' of data files are identical to any other file type so if you have mastered locating files from the directory entries, you will not have trouble in this department either.

10.8.1 'ASCII' DATA FILES

ASCII data files are the easiest to read. Everything is 'readable' and will display with the 'DD' function of "SUPERZAP". Figure 10.4 is a typical sector of an ASCII FILE.

(figure 10.4)

```

30E500 2031 3030 3020 2020 2020 2020 2020 2020 .1000.....
30E510 3230 3734 4E2D 3736 2C47 454D 2020 2043 2074N-76,GEN...C
30E520 4F52 504F 5241 5449 4F4E 2C32 3232 2057 ORPORATION,222.W
30E530 494C 5348 4952 4520 424C 5644 2E2C 4C4F ILSHIRE.BLVD.,LO
30E540 5320 414E 4745 4C45 532C 4341 4C49 464F S.ANGELES,CALIFO
30E550 524E 4941 2C39 3030 3137 2C48 4152 5259 RNIA,90017,HARRY
30E560 2048 2E20 4448 4F52 4520 2020 2020 2020 .H..DHORE.....
30E570 2041 2E46 2E47 2E41 2C20 4841 4E4B 2044 .A.F.G.A.,HANK.D
30E580 2E20 504F 4E42 414D 2020 472E 472E 2020 ..PONBAM..C.G...
30E590 412E 462E 472E 412C 5245 5345 5256 4544 A.F.G.A,RESERVED
30E5A0 2C49 2044 4944 4E27 5420 5448 494E 4B20 ,I.DIDN'T.THINK.
30E5B0 594F 5520 574F 554C 4420 4649 4E44 2054 YOU.WOULD.FIND.T
30E5C0 4849 532C 0D46 5241 4E4B 2C0D 322E 3020 HIS,.FRANK,.2.0.
30E5D0 2030 362F 3031 2F37 390D 2020 2E2E 2E20 .06/01/79.....
30E5E0 482E 502E 2020 0D2C 2C5E E5E5 E5E5 E5E5 H.P.,,,,,,
30E5F0 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 E5E5 .....

```

You will notice that each successive data item of an ASCII file is separated by a ',' and is represented in the HEX portion of the display as a HEXADECIMAL '2C'. The 'EOF' byte of an ASCII file is represented by an 'empty data item'. i.e., two commas with nothing in between (Relative bytes 'E7' and 'E8').

To repair data all you must do is "ZAP" the necessary bytes with the ASCII values that represent the data you wish to change. For instance, suppose you want to change the '1000' in the first line of the above display to '2000'. The numeral '1' is ASCII CODE '31' (HEX). Change the '31' at relative byte '01' to '32'. Right before your amazed eyes the '1000' will become a '2000'.

I would like to caution you Stanley Rifkin fans that Stanley didn't do so well in the 'getaway' department, so if you have visions of doctoring-up a database on a payroll program, forget it.

10.8.2 'RANDOM' DATA FILES

Random data files are a little more trouble to alter. However, an enterprising soul such as yourself will find it not too difficult. The mere fact that you've read this far is commentary on your tenacity.

The following (figure 10.5) is a sector from a random file. You will notice that there are no 'delimiters' in the file, such as commas or carriage returns. Each 'data item' is butted together. The separate data items are separated, in your program, with the 'FIELD' statement.

The numeric data, such as 'INTEGER', 'SINGLE PRECISION', and 'DOUBLE PRECISION' numbers are represented in compressed binary format. To effectively work on the data in a 'RANDOM' file, first make a 'MAP' of your data as it will be on each sector. Use one of the 'MAP's in chapter 6 as a guide. Once you have data mapped out, it is a simple matter to modify each data item.

(figure 10.5)

```

F01700  5045 4143 4850 4954 2047 494E 4745 5220 PEACHPIT.GINGER.
F01710  3131 3530 2054 454E 4E59 534F 4E20 2332 1150.TENNYSON.#2
F01720  3420 2020 2020 2020 004D 3091 5543 4A54 4.....M0.UCJT
F01730  0000 0000 2D86 E000 3086 8000 2020 2020 .....-....0.....
F01740  2020 2020 2020 2020 2020 2020 2020 2020 .....
F01750  2020 2020 2020 2020 2020 2020 2020 2020 .....
F01760  0000 0000 2045 86F0 0020 4B86 8800 2020 .....E....K.....
F01770  2020 2020 2020 0000 0000 2020 2020 2050 .....P
F01780  4F4C 4543 4154 2052 5554 4820 3131 3636 OLECAT.RUTH.1166
F01790  3120 4B49 4F57 4120 4156 452E 2023 3420 1.KIOWA.AVE..#4.
F017A0  2020 2020 2020 2080 E02F 9155 4344 4300 ...../.UCDC.
F017B0  0000 002D 86E0 0030 8680 0020 2020 2020 ....-....0.....
F017C0  2020 2020 2020 2020 2020 2020 2020 2020 .....
F017D0  2020 2020 2020 2020 2020 2020 2020 2000 .....
F017E0  0000 0020 4586 F000 204B 8688 0020 2020 .....E....K.....
F017F0  2020 2020 2000 0000 0020 2020 2020 0000 .....

```

If you study the above figure closely, you will find that there are two identical sub-record layouts on this sector. Now hark back to your Radio Shack Disk Manual, and you will find in the rather obscurely described sections on 'random I/O' that records may not span sectors and that there is something about a 'PHYSICAL RECORD' and a 'LOGICAL RECORD'. A sector, such as the one above, is a 'PHYSICAL RECORD'. Each sub-record in that 'PHYSICAL RECORD', such as the two above, are 'LOGICAL RECORDS'.

There is one slight difference in the way various disk operating systems configure random files. TRSDOS 2.1 and NEW DOS 2.1 only permit 255 byte random file records. DO NOT CHANGE RELATIVE BYTE 'FF' ON DATA RECORDS THAT ARE ACCESSED WITH THESE OPERATING SYSTEMS.

TRSDOS 2.2, VTOS 3.0 and Apparat's new SUPER DOS 1.0, all permit the use of 256 byte records. SUPER DOS will even permit records as large as 4095 bytes in a single record! (I'm not supposed to talk about that yet, but I figured that you needed the information. Besides, I would like to drive the Radio Shack software development people crazy wondering how they did THAT!)

Let's take a closer look at figure 10.5 before we go to the next chapter. The first 40 bytes contain a name and street address. (But I thought you said numerical data was represented in HEX format?) I did. I did. But it may also be represented in ASCII fashion IF you fielded your input as a string without converting it to one of the numerical data types.

The next 4 bytes represent a 'SINGLE PRECISION' zip code. The zipcode, for this first sub-record is '90266'. To add confusion to the obscure, 90266 (DECIMAL) is equal to '01609A' (HEX). On the file however, it is represented as '004D3091' (HEX)! Now I ask you, "Does that make sense?" Yes, as a matter of fact it does.

You must first understand how numbers are represented internally, to make complete sense out of the various data types. BASIC's number crunching routines require that the sign, exponent and floating point representation of the number be stored as well as the actual number. This will be the subject of another book (Working Title: BASIC COMMENTED, LISTED AND NARRATED.).

In the meantime all you really need to know is what your numbers look like. Here is a way to decode HEXADECIMAL representation of the various data types from BASIC itself. Enter the following program in BASIC and run it.

```

100 A = 90266           : ' SET VALUE OF 'A'
110 A$ = MKI$(A)         : ' CONVERT TO STRING
                           REPRESENTATION
120 PRINT A$             : ' LOOK AT A$
130 PRINT LEN(A$)        : ' LOOK AT LENGTH OF A$
140 FOR X = 1 TO LEN(A$) : ' SET LOOP
150 PRINT ASC(MID$(A$,X,1)) : ' LOOK AT DECIMAL VALUES
                           OF A$
160 NEXT                 : ' LOOP
<RUN>

```

PROGRAM DISPLAY	MEANING OF DISPLAY
M0	← Display representation of A\$
4	← Length of A\$
0	← 1st ASCII character = 00 (HEX)
77	← 2nd ASCII character = 4D (HEX)
48	← 3rd ASCII character = 30 (HEX)
145	← 4th ASCII character = 91 (HEX)

To see how other data types are represented substitute 'MKI\$' with one of the following:

```

MKI$ -- Converts INTEGER to string representation.
MKS$ -- Converts SINGLE PRECISION to string representation.
MKD$ -- Converts DOUBLE PRECISION to string representation.

```

By reviewing the LEVEL II manual and the DISK manual you may learn more about data types.

Now, assuming that you have tried the BASIC program, to better understand how your various data types are represented and have made a map of the random data file you wish to "ZAP", you're ready to go to work. Good luck.

10.9 RECOVERING A LOST TENNIS BALL

Look under the Volkswagen or in the neighbors' ivy.

11.0 RECOVERING 'ELECTRIC PENCIL' ERRORS

Without Michael Shrayer's ELECTRIC PENCIL, this book would not have been possible. I have used every feature of the program and if the TRS-80 were used for nothing else except word processing, this program would justify the entire hardware cost. Unfortunately, the 'PENCIL' does a few peculiar things ... some of them are the fault of the program and others are a result of the operator.

Since 'PENCIL' is so widely used, I thought it would be a good idea to address some of the data recovery techniques that may be used on ELECTRIC PENCIL. Needless to say, this chapter (as well as the others) is the result of having had to recover more than one or two occasional errors.

11.1 RECOVERING 'ELECTRIC PENCIL DOS ERROR 22'

I don't know where 'ELECTRIC PENCIL' gets its error codes but when 'PENCIL' gives you 'DOS ERROR 22' it is NOT error '22'. The book says DOS ERROR 22 is a 'HIT' sector error. (Michael, please pay attention. There is going to be a test on this tomorrow!)

The error in this case is a wrong sector count in the 'FPDE' (BYTES 14 and 15), or the 'EOF' byte (BYTE 3) is wrong. Whatever the values in these bytes are, change them by ADDING at least '1' to either or both of these values. You may make these bytes ANYTHING you want as long as they are greater than the values they should be! For instance, figure 11.1 is an example of an ELECTRIC PENCIL file directory entry with a 'DOS ERROR 22' error.

(figure 11.1)

RELATIVE BYTE 3

111340	1000	0074	0045	5252	3232	2020	2050	434CERR22...PCL
111350	9642	9642	0900	2401	FFFF	FFFF	FFFF	FFFF	.B.B..\$.....

RELATIVE BYTES 14 & 15

The diagram shows a file directory entry with two lines of data. The first line is labeled 'RELATIVE BYTE 3' and the second line is labeled 'RELATIVE BYTES 14 & 15'. The data is presented in a table format with columns for relative byte addresses and their corresponding values. The first line shows a 'DOS ERROR 22' and the second line shows a file entry with a sector count error.

The most likely thing that is wrong with the file is that BYTES 14 & 15 are incorrect. Since there is also the possibility that BYTE '3' may also be incorrect, "ZAP" both locations. Then, load the file into 'PENCIL' and 'SAVE' it. If you go back and look at it, you will find that it will have corrected itself and the proper values will have been inserted into the offending bytes. Figure 11.2 is an example of the "ZAP"s necessary to correct the 'DOS ERROR 22'.

Be sure and make this value LARGER than your file actually is. 'FF' is 255 DECIMAL, and will cover most situations.

```

      :
    -:
111340 1000 00FF 0045 5252 3232 2020 2050 434C .....ERR22...PCL
111350 9642 9642 FF00 2401 FFFF FFFF FFFF FFFF .B.B..$.....
      :
    -:

```

"ZAP" this with 'FF' too.

11.2 'LOST' ELECTRIC PENCIL ON DISK ('OVER WRITTEN' FILE)

There are two reasons for this unfortunate circumstance:

UNFORTUNATE REASON #1: You were working on this file only a couple of days ago. Everything was working smoothly, and when you were through entering your text, you saved the file to the disk. You removed the disks from the drives, shut everything down and went home (or to another room) and watched an exciting rerun of I LOVE LUCY before dinner.

Several days (or hours) later you went back to the computer to use that file. You bring up your 'ELECTRIC PENCIL' program and load the file. What?!!!! It's GONE! There are only 3 carriage returns on the screen! After the blood returns to your brain, and you finally begin to believe your eyes, reason returns to your fogged brain; you decide you must have saved it on another disk.

Forty-seven disks later you give up and say to yourself, "...dammit, I KNOW I saved that file. I wonder what could have happened to it? It must have been eaten-up by the machine or something." Thus, you conclude that there are mysteries that are beyond human understanding and consult the TV Guide to see what time Mork & Mindy come on.

The truth of the matter is that nobody ate nothin'. Everything worked exactly like it was supposed to, you screwed up. In your dazed and confused state, after typing for 6 hours, you 'SAVE'd your file WITH THE CURSOR AT THE END INSTEAD OF AT THE BEGINNING OF THE FILE!

UNFORTUNATE REASON # 2: You accidentally 'KILL'ed the file by using the wrong file name. I don't know why you did it but it sometimes does happen.

RECOVERING UNFORTUNATE REASON # 2, refer to 9.2, "RECOVERING A KILLED FILE". To recover UNFORTUNATE REASON # 1 simply follow the procedures below. (See Chapter 9.7 for details on 'ELECTRIC PENCIL' files.)

1. Find the 'EOF' BYTE in the file.

2. Change the 'EOF' byte to any valid ASCII character ('20' or '0D' works nicely)
3. "ZAP" the directory 'EOF' byte with an 'FF' ('FPDE' BYTE 3).
4. "ZAP" the sector count bytes ('FPDE' BYTES' 14 & 15) with a HEX value larger than the actual sector count - 'FF' will work here also, in most cases.

Since the old 'EOF' marker is still in the file you won't have to worry about where it is or where to put it. Just go to 'PENCIL' and load the file. If you get a 'DOS ERROR 22' you didn't make the sector count byte large enough.

11.3 RECOVERING A 'LOST' ELECTRIC PENCIL FILE IN MEMORY

I know this has nothing to do with the disk, but before you can recover something on disk, you have to get it there. I have had occasions when 'PENCIL' does its outer space trick and have had desperate need to know how to get it back so I could get it onto the disk in the first place!

Here is the picture:

You are inputting text into 'PENCIL' and all of a sudden the machine 'BOOT's or you have put in a particularly long line; you hit <ENTER>; the screen goes 'funny' and suddenly funny little characters appear on the screen. It might be described as what Android Nim would look like after swallowing a hand grenade.

Here is your recovery procedure:

1. Stop cursing. You cannot be heard in Fort Worth or by Shlayer.
2. Type: <CONTROL> 'O' and get into DOS.
3. Type: DEBUG <ENTER>
4. Hit the 'BREAK' key. You will now enter 'DEBUG'.
5. Type: G5C61 <SPACE> or <ENTER>
6. THERE IT IS! IT'S BACK!
7. Immediately save your file to the disk. DO NOT HIT 'BREAK' TO EXIT THE SUB-COMMAND TABLE or you will re-enter debug.
8. Hit 'RIGHT ARROW' to exit the SUB-COMMAND TABLE.
9. If the screen 'went funny' before you 'lost' your file, enter the search and replace function. (<COMMAND> 'V').
11. Replace the line you were working on before it 'went funny', with something shorter. If you don't remember what you were working on, exit 'PENCIL' and fix the file with "SUPERZAP" by putting carriage returns or spaces ('0D' or '20') in the line that is 'too long'.

11.4 RECOVERING DISK FILE WON'T LOAD ('FILE AREA FULL' ERROR.)

This isn't an 'error', in the true sense of the word. What has happened is that you typed a large file into memory. You saved the text to disk. At some later time, you couldn't load the file because every time you tried, you got the message: 'FILE AREA FULL'.

'PENCIL' will allow you to 'SAVE' a file that is larger than you can 'LOAD'. (Now isn't THAT nice, Ollie?) Yes indeed, just one more little thing to make life interesting. All is not lost. In fact none of it is lost. All we have to do is break up the file into smaller segments and it will load just fine.

Create an 'FPDE' by saving a one word 'dummy' file while in 'PENCIL'. Use a file name that you would normally use anyway since there is no reason to 'SAVE' it again with another name.

Now, go to the last few sectors of the file that won't load and copy those sectors to the EXTENT FILE AREA pointed to by the 'dummy' file name. "ZAP" the 'dummy' file 'FPDE' BYTE 3 and BYTE 14 with 'FF', just like we did in recovering a 'DOS ERROR 22'. Now that portion of the file will load. 'LOAD' it and 'SAVE' it back and everything will take care of itself.

Next we have to fix the original file so it won't try to load the whole thing. Go back to that first sector that we moved to the 'dummy' file... (AH-HA! You forgot which one it was and you didn't take notes, did you? See how important taking notes can be?) ..."ZAP" a '00' anywhere in the sector and that will take care of that portion of the file. Now both segments of the file will 'LOAD' and you're on your way again.

11.5 ELECTRIC PENCIL GOODIES

Here are a couple of things that might make your day brighter, for what it's worth.

To make 'ELECTRIC PENCIL' compatible with NEWDOS 2.1 all we have to do is change 3 bytes in relative sector 0 of 'PENCIL' to '00 00 00'. Find relative sector 0 then, at or near relative byte 'AE', you will see the following code:

F332 9B46 C36F

"ZAP" this so it reads:

F300 0000 C36F

Another thing you might like to do is speed-up 'PENCIL's cursor - a simple one byte change. In relative sector 10 (HEX) on or about relative byte '7B' you will see the following code:

0600 10FE 1116

"ZAP" this so it reads:

0664 10FE 1116

My cursor is set at '50'. The '00' that is in there now, is a value of 256 - this is as SLOW as it can possibly go. A little experimentation will tell you what value to put into this byte. A word of caution ... '50' really makes that thing zip along.

In addition to all the before mentioned 'goodies' you can do with the 'PENCIL', here are a few more.

WRITE BASIC PROGRAMS IN 'PENCIL'. Wouldn't it be neat to be able to write programs in BASIC and have the editing features of 'PENCIL'? It's not only possible but I do it all the time. In the appendix there is a BASIC program called 'SEARCH'. It was written in 'PENCIL', and documented in 'PENCIL'.

There is no secret, all you have to do is just do it. No tricks, no special things to do, just write the program like you would normally do, only use 'ELECTRIC PENCIL' to write the program. When finished, write the text to disk; exit 'PENCIL', go to BASIC, and 'LOAD' and 'RUN'. There are only two things to watch for. (1) Your filename will have '/PCL' on the end of it and (2), ONLY PUT CARRIAGE RETURNS AT THE END OF A STATEMENT LINE. Now go do it and see how easy it is.

LOAD A PROGRAM WRITTEN IN BASIC INTO PENCIL FOR EDITING. Have you ever wished you could change all those 'PRINT's to 'LPRINT's in one swell foop? Not hard at all, once you have your BASIC program loaded into 'PENCIL' for editing. Here is all you do. (1) Make sure there are no lines longer than 30 characters without spaces. If you pack (cram) your statements together, open 'em up here and there. (2) Save the program in ASCII mode with a filename that includes '/PCL' for a file name extension. After it's saved, "ZAP" '00' into the last byte of the program file. There it is. You are ready to load it into 'PENCIL'.

WRITE AN ASSEMBLER, OR FORTRAN PROGRAM IN PENCIL. If you have 'MACRO-80' all you have to do is write your program in 'PENCIL' with or without line numbers. 'SAVE' the text file as-per-usual then exit 'PENCIL', run MACRO-80 ('EDIT') and load the 'ELECTRIC PENCIL' file using 'Mac80' commands. 'Mac80' will append the line numbers and will give you the option of saving the source with or without the line numbers. Assemble or compile and away you go.



12.0 CORRECTING THE 'GAT' AND 'HIT' SECTORS

'GAT' errors can be particularly disastrous. TRS-DOS 2.1 will occasionally de-allocate GRANULES. For those of you who are technically minded, I will quote from the APPARAT documentation, describing the cause of this disaster:

'CLOSE' in 'SYS3/SYS' causes a major system disaster when it releases an 'FXDE' by not preserving the contents of the CPU register DE, which contains a count of +1, of the sectors yet to be freed, when freeing a no longer needed 'FXDE'.

This error is compounded by the branch at '4ED9' by not implicitly ending deallocation of GRANULES, when the file is known to have no more GRANULES assigned.

These errors cause all 'write'able main memory for 3000 - 42XX (HEX) to be set equal to 'FF', where 'XX' is the relative position within the sector of the last byte of the 'FPDE' pointed to by the last 'FXDE' that was released.

The corresponding sector in the directory is also filled with 'FF' to that relative point. As that continues, the 'GAT DIRECTORY SECTOR' is modified to free up GRANULES at random in tracks '00' through 'FF', with most tracks below 80 hex.

If this continues to go undetected, this will cause GRANULES previously allocated to other files, to be allocated again in subsequent file allocations! This includes reallocation of 'BOOT/SYS' and 'DIR/SYS' GRANULES, eventually clobbering them.

Now, if that's not bad enough, read on. Files whose 'FPDE' preceded the destroyed 'FPDE', in the 'DIRECTORY ENTRY SECTOR' will disappear from the system and if a file's 'FXDE' was so destroyed, you will have horrendous trouble and should be considered lucky if TRS-DOS even detects an error!

Almost as bad, CPU register 'HL' is not decremented to the 2nd byte of the next lower EXTENT nor is it protected by the directory sector write call at '4F08'. This causes the two bytes (whatever they are, at the time) at '41FF' and '4200' to be used as the next lower EXTENT for the file, causing a somewhat random deallocation of GRANULES, usually in the range of tracks '00' to '10' hex.

And still more! If a new 'FXDE' is allocated to the file and then if the diskette is found to be full, 'SYS3/SYS' malfunctions (at some future time) when 'CLOSE' tries to free the space assigned to that 'FXDE'. It assumes there is some 'FREE' space when there is none!

You will have to pardon me while I do a little preaching. Would you, if you were an international distributor of 'quality software', sell and distribute software with KNOWN disastrous errors and not tell your users? Would you cover up your errors by simply not telling your users that the errors existed and that IF they had a problem, it was most likely the fault of the hardware? I wouldn't do that. I don't think you would either. (The lawyers who looked at this manuscript for libelous statements wouldn't do that.) Good grief, WHO would do that?

It's damn difficult to write and release bug-free software and there are excellent software packages that contain bugs but the authors are burning the midnight oil to correct them and warn their users... while solutions are being sought. Now, WHO would turn-out

software and not admit that there is something wrong? (Sure beats hell out me, lieutenant, I'm not the regular crew-chief.)

Why are we are treated like mushrooms (kept in the dark and fed B.S.!) and told that certain aspects of certain programs were not "...fully implemented"?

Enough of this grousing.... continue reading this saga...

If this 'GAT' sector problem is detected soon enough, very little damage will occur. The above described 'bug' (this one is so big, it could be used in a Japanese horror movie suitable for showing on Channel 13 at 3 A.M.) will also explain how files get into other files. With the deallocated GRANULES, the DOS thinks that it is OK to store something to a GRANULE already being used for another file. Then when you attempt to 'LOAD' or 'RUN' what you think is file 'A' you get 'B' instead.

12.1 THE 'GAT' FIX

- (1) Using the 'DIRCHECK' utility of NEW DOS+ list the directory and note any errors that may have accumulated in the 'GAT' and 'HIT' sectors. See section 3.1 for details on 'DIRCHECK'.
- (2) If you do not have NEW DOS+ you will have to go through each 'FPDE/FXDE' entry in the directory sectors, note the EXTENT addresses and GRANULE counts and then compare these to each GRANULE track by track. (Sorry, there isn't a faster way that I know of.)
- (3) "ZAP" each offending GRANULE with the correct allocation. (Also see figure 6.7 for allocation codes.)

```
***** CAUTION *** CAUTION *** CAUTION *****
**
** Be sure to 'KILL' extraneous files using **
** the same 'GAT' sectors as 'good' files. **
** Failure to do so will cause additional **
** errors to occur. **
**
*****
```

12.2 THE 'HIT' FIX

Basically, this is the same procedure as above. How to read 'DIRCHECK's error list, regarding the 'HIT' sector is also discussed in 3.1, above. Finding bad 'HIT' sector bytes is a little easier than finding bad 'GAT' bytes.

There should be as many 'HASH' codes (non-zero bytes) in the 'HIT' sector as there are active files. Every file that displays with 'DIR' AND LOADS or is accessible with an 'OPEN' statement has a valid 'HASH' code. Failure to do ONE of these things is an indication that you are about to have or are having problems with the disk.

How to obtain the correct 'HASH CODE', for a file name, is covered in 10.1 above.

13.0 SOME THINGS YOU CAN DO

It always helps to have someone point out some new directions - open up our imagination, so to speak. What I'm attempting to do in this chapter is give you some ideas that will hopefully cause you to have some more ideas on your own. The limitations you will encounter on the computer are almost entirely of your own making. Adopt the philosophy that "there is a way" and sooner or later, you'll find it.

13.1 CONSTRUCTING 'ELECTRIC PENCIL' FILES IN BASIC.

This is so easy you'll wonder why it never occurred to you before. It was only after looking at 'PENCIL' files, with "SUPERZAP" that it dawned on me that these files were almost ordinary ASCII files. With a little experimentation and the use of "SUPERZAP", I was able to figure out everything I needed to know. Try this experiment.

LOAD BASIC and enter the following program:

```
100 CLEAR: CLS           clear the stack; clear the screen
110 OPEN"O",1,"TESTONE/PCL" set-up filename with 'PENCIL' extension
120 A$="THIS IS A TEST    initialize 'A$' with text
    WHICH WILL BUILD A
    PENCIL FILE IN BASIC"
130 PRINT #1, A$         write 'A$' to the file
140 PRINT #1, CHR$(0)     insert the 'EOF' marker for
                          'ELECTRIC PENCIL' ('00' HEX)
150 CLOSE                close the file
<RUN>                   run the program
```

Of course, this program is very simple and I realize that it could have been written in a much more sophisticated style but it is very easy to 'see' how it functions. Now load "PENCIL" and load 'TESTONE'.

13.2 'LOAD'ING A BASIC PROGRAM OR ASCII DATA FILE INTO 'ELECTRIC PENCIL'

You will only have difficulty in doing this, if you have 'packed' your BASIC program or data file, i.e., eliminated all spaces between words, statements, and characters.

One of the really neat things about using 'PENCIL' with a BASIC file as text, is the global search and replace. You can replace every single 'PRINT' with an 'LPRINT' in less than a couple of seconds! You can also use it to make translations from one dialect of BASIC to another. Using 'PENCIL' enter the text for a BASIC program out of a magazine. Don't try to make all of the statement conversions. At the end of the magazine version, enter the subroutines that replace the non-'RUN'able statements. Now 'SEARCH AND REPLACE' these statements with a 'GOSUB' to your subroutine. In a few minutes you can make a translation that would normally take hours or even days!

'PENCIL' MUST have at least one space every 30 or so characters for its video display management routine. Now that you know what can go wrong, let's give it a whirl.

Enter the above program just as it is typed; but when you 'SAVE' it, use this or a similar name:

```
SAVE"FILETEST/PCL",A
```

Remember, 'ELECTRIC PENCIL' only loads files with the 'filename extension' of '/PCL'. The ',A' at the end of the 'SAVE' statement, will cause the program to be 'SAVE'd in ASCII format.

Now, with "SUPERZAP", locate the end of the "FILETEST/PCL" file and "ZAP" the last carriage return ('0D' HEX) with a '00'. Execute 'PENCIL' and load 'FILETEST'.

Another way to do the same thing, if you are using NEW DOS, is to use the 'OPEN"E"' function. After you have 'SAVE'd the program, type 'NEW' and enter and run the following:

```
100 OPEN"E",1,"FILETEST/PCL"  
200 PRINT #1, CHR$(0)+" "  
300 CLOSE  
<RUN>
```

This will open the file at the end and write the 'EOF' marker for 'ELECTRIC PENCIL'. The blanks between quotes will guarantee that the file will load into 'PENCIL' and not give you that 'DOS ERROR 22' crap. You may do nearly the same thing with TRSDOS except you will have to read the file and write it to another file, then when you get to the end of the original file, write the 'EOF' marker to the new file. Actually, you should save yourself a lot of grief and aggravation, get NEWDOS and be done with it!

The above techniques can be used with data files as well as program files.

13.3 MAKING 'PENCIL' FILES INTO BASIC FILES.

Actually you don't have to do anything except enter your program into 'PENCIL'. Save it to disk, and run it. Don't forget to use the '/PCL' file name extension when calling your program from BASIC.

A 'PENCIL' file is an ASCII file. It will load into BASIC just like any other BASIC file 'SAVE'd with the ASCII option. You must remember that if you 'SAVE' the file back, while in BASIC, that the 'EOF' marker for 'PENCIL' will not be there and the file will no longer load into 'PENCIL'. Also see 13.2 above.

13.4 CONVERTING 'DATA TYPES' IN RANDOM FILES

In chapter 9.8.2 we discussed a method of repairing data files. There is also a short BASIC program in that chapter, that converts 'random numerical data' into its ASCII equivalent.

To convert a data type, all that is necessary is to have the proper information in the right place AND to re-code your 'FIELD' statements.

Suppose that you had a double precision number beginning at say, relative byte 'A7'. Your lightning-quick-bear-trap-mind will immediately recall that a double precision number is eight bytes long.

Your problem, Mr. Phelps, is to convert that to a single precision number field. First obtain the single precision string contents, using the BASIC program in 9.8.2. Convert the DECIMAL values to HEX. "ZAP" the four HEX values into the appropriate sector, beginning at relative byte 'A7'. "ZAP" '2020 2020' into the remaining (and now unused) four bytes (or anything that is appropriate for your file).

Next change the field statement so that only 4 bytes are fielded for the new single precision number and 4 bytes for the unused 4 bytes (or whatever you have converted those 4 bytes to.)

The last thing is to change the 'MKD\$' statement and the 'CVD' statements to 'MKS\$' and 'CVS'. Now go!

13.5 CONVERTING DATA IN ASCII FILES

Run "SUPERZAP" and using the 'MODnn' function, type in whatever you want. Be sure to use commas for delimiters or you will get a few more characters than you bargained for into the wrong string. That's it.

13.6 MAKING BASIC PROGRAMS 'UNLISTABLE'

There is no such thing as total protection. This will make a program 'unlistable' as long as the user never reads this book or figures things out for himself.

Save a BASIC program that contains a dummy string like this:

```
DU$="*****"
```

Using "SUPERZAP", find that string in the program, as stored on the disk. The HEX code for '*' is '2A'. "ZAP" those '2A's with '1228 1212 1212 1212' etc. Now load and list the program. 'LLIST' the program. Lots of paper, huh?

If you will consult Appendix C/1 of the LEVEL II manual you will find the 31 (DECIMAL) ASCII control codes. Try placing different codes into the string and see what happens when you try to 'LLIST' or 'LIST' the program.

13.7 ADDING COMMANDS TO SUPERZAP

"SUPERZAP" is a very well written BASIC program and is easy to make modifications to. I myself, have a constant need to run 'DIRCHECK' while I'm still in "SUPERZAP" - especially if I'm making corrections to the 'GAT' or 'HIT' sectors. Of course you can type: <BREAK>, 'CMD"DIRCHECK', answer the prompts and then when the program is through and returns you to BASIC (IF you're using NEWDOS), type, 'CONT', 'X' and then re-enter "SUPERZAP" where you left off.

My particular version of "SUPERZAP" has had 'UP-ARROW' added as one of the commands that functions while in 'DD' mode that automatically runs 'DIRCHECK'. Load "SUPERZAP" and enter the program lines below. Try it out by using the 'DD' function. While in 'DD' and when a sector is being displayed, hit 'UP-ARROW'. MAGIC!

When 'DIRCHECK' is all through, you are returned to the exact place you've left off.

(SUPERZAP 2.0)

```
2210 IF A$="↑" THEN 60000
```

```
60000 CLS: PRINT@345,"DIRCHECK"
```

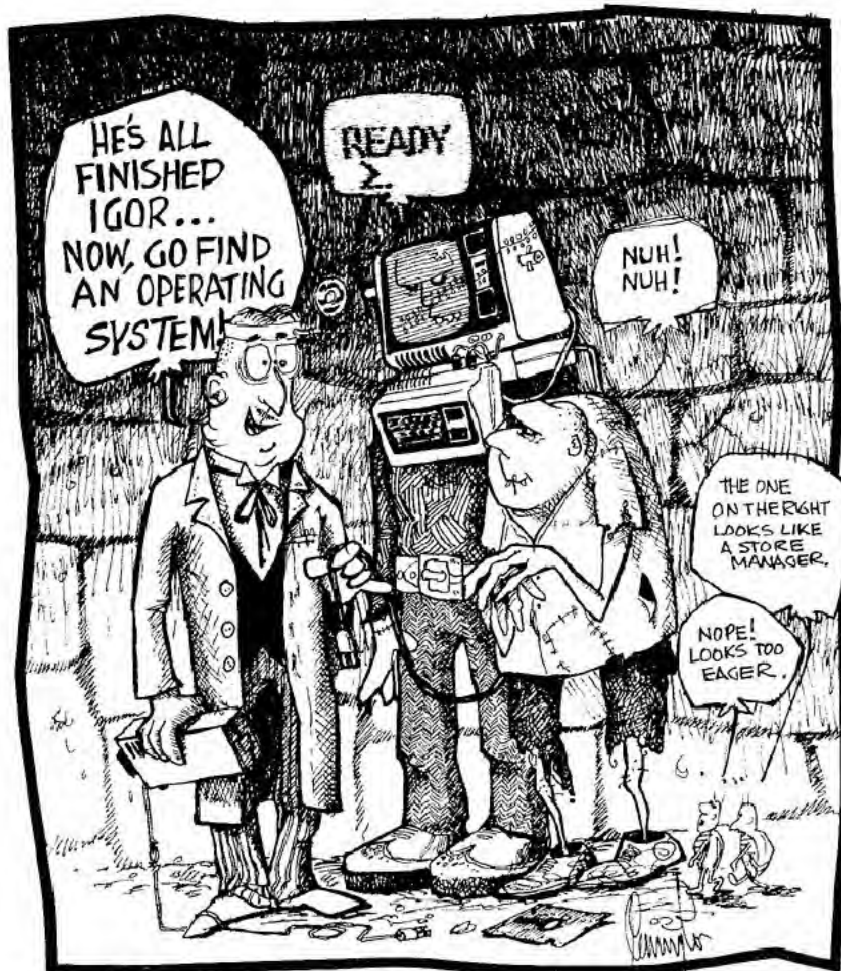
```
60010 CMD"DIRCHECK":A$="R":GOTO 2210
```

If you like the change, 'SAVE' the program back to disk. If you think of any commands you would like to add, use the same technique as I have used and add your own special commands.

13.8 READING THE 'DIRECTORY' FROM BASIC

If you have NEWDOS try opening a file with the file name: DIR/SYS, and read the first record into a random buffer fielded as follows: FIELD 1, 255 AS A\$

You will get an error message on the 'GET' statement but try printing out your string anyway. What? you say it worked? My word, amazing isn't it. Simply trap the error, so your program does not 'crash' and continue along your way. With this little trick you can have your BASIC program read your disk directories.



APPENDIX A

APPENDIX A

I GLOSSARY

ACCESS The operation of seeking, reading or writing data on a storage unit (in this case, the diskette).

ACCESS TIME The time that elapses between any instruction being given to access some data and that data becoming available for use.

ACTIVE RECORDS TABLE (ART) A table of binary values in which the relative position of a single value determines the status of a record with the same relative position; i.e., the Nth binary number determines the status of the Nth record. **EXAMPLE:** If the 8th binary number in the table is a zero, then the 8th record is inactive. Conversely, if the 8th binary number in the table is a one, then the 8th record is active.

ADDRESS An identification (number, name, or label) for a location in which data is stored.

ALGORITHM A computational procedure.

ALPHANUMERIC (CHARACTERS) A generic term for numeric digits, alphabetic characters, punctuation characters and special characters.

ALPHANUMERIC STRING A group of characters which may include digits, alphabetic characters, punctuation characters and special characters, and may include spaces. (NOTE: a space is a 'character' to the computer, as it must generate a code for spaces as well as symbols.)

ASCII Abbreviation for American Standard Code for Information Interchange. Pronounced: Ass-KEY. Usually refers to a standard method of encoding letter, numeral, symbol and special function characters, as used by the computer industry.

ASSEMBLY LANGUAGE A machine level language for programming, such as Radio Shack's "EDITOR/ASSEMBLER" which uses Z-80 processor mnemonics and automatically 'assembles' machine readable code from the mnemonics.

BASE A quantity of characters for use in each of the digital positions of a numbering system.

BASE 2 The 'BINARY' numbering system consisting of more than one symbol, representing a sum, in which the individual quantity represented by each figure is based on a multiple of 2.

BASE 10 The 'DECIMAL' numbering system - consisting of more than one symbol, representing a sum, in which the individual quantity represented by each symbol is based on a multiple of 10.

BASE 16 The 'HEXADECIMAL' numbering system - consisting of more than one symbol representing a sum, in which the individual quantity represented by each symbol is based on a multiple of 16.

APPENDIX A

BINARY See 'BASE 2'

BIT A single 'BINARY' digit whose value is 'zero' or 'one'.

BOOLEAN A form of algebra applied to binary numbers which is similar in form to ordinary algebra. It is especially useful for logical analysis of binary numbers as used in computers.

'BOOT' - BOOTSTRAP A machine language program file that is put onto every diskette by the 'FORMAT' routine. This routine is invoked when reset or power-on occurs. It automatically loads the necessary programs (SYS0/SYS) to cause the computer to respond to the DOS commands; i.e., the machine is 'BOOTSTRAPPED' or 'BOOTED' into operation.

BUFFER A small area of memory used for the temporary storage of data to be processed.

BUFFER TRACK A track on a diskette used for the temporary storage of data or program material during a recovery process.

BUG A software fault that results in the malfunction of a program. May also refer to hardware malfunctions.

BYTE Eight 'BITS'. A 'BYTE' may represent any numerical value between '0' and '255'.

CLOBBERED A slang term referring to the non-operation of software, hardware, computer device, or storage media (such as disk) usually as the result of a program or hardware error.

COMMAND FILE A file consisting of a list of commands, to be executed in sequence.

CONTIGUOUS Adjacent or adjoining.

CONTROL CODE In programming, instructions which determine conditional jumps are often referred to as control instructions and the time sequence of execution of instructions is called the flow of control.

CRC ERROR Cyclic Redundancy Check. A means of checking for errors by using redundant information used primarily to check disk I/O on the TRS-80.

DATA BASE A collection of interrelated data stored together with controlled redundancy to serve one or more applications. The data are stored so that they are independent of programs which use the data. A common and controlled approach is used in adding new data and in modifying and retrieving existing data within a data base. A system is said to contain a collection of data-based information if they are disjoint in structure.

APPENDIX A

DATA BASE MANAGEMENT SYSTEM The collection of software required for using a data base.

DATA ELEMENT Synonymous with 'DATA ITEM' or 'FIELD'

DATA TYPE The form in which data is stored; i.e., integer, single precision, double precision, 'alphanumeric' character strings or 'strings'.

DEC Initials for Directory Entry Code.

DECIMAL See 'BASE 10'.

DIRECT ACCESS Retrieval or storage of data by a reference to its location on a disk, rather than relative to the previously retrieved or stored data.

DIRECT STATEMENT (IN FILE) A program statement that exists in the disk file that is not assigned a line number.

DIRECTORY A table giving the relationships between items of data. Sometimes a table or an index giving the addresses of data.

DISPLACEMENT A specified number of sectors, at the top or beginning of the file, in which the 'bookkeeping' and file parameters are stored for later use by the various program modules.

DISTRIBUTED FREE SPACE Space left empty at intervals in a data layout to permit the possible insertion of new data.

DOUBLE PRECISION A positive or negative numeric value, 16 digits in length, not including a decimal point (EXAMPLE: 99999999999999.99).

DUMP To transfer all or part of the contents of one section of computer memory or disk into another section, or to some other computer device.

DYNAMIC STORAGE ALLOCATION The allocation of storage space by a procedure based on the instantaneous or actual demand for storage space by that procedure, rather than allocating storage space to a procedure based on its anticipated or predicted demand.

EATEN (DIRECTORY/DISK) Slang term. See 'CLOBBERED'.

EMBEDDED POINTERS Pointers in the data records rather than in a directory.

ENTITY Something about which data is recorded.

EOF Initials for 'END OF FILE'. It is common practice to say that the EOF is record number nn or that the EOF is byte 15 of sector 12. Hence, it is a convenient term to use in describing the location of the last record or last byte in a file.

APPENDIX A

EXTENT A contiguous area of data storage.

FILE A collection of related records treated as a unit; The word file is used in the general sense to mean any collection of informational items similar to one another in purpose, form and content.

FILE PARAMETERS The data that describes or defines the structure of the file.

FILESPEC A file specification and may include the 'FILE NAME', 'FILE NAME EXTENSION', 'PASSWORD', and 'DISK DRIVE' specification.

FIELD See 'DATA ITEM'.

FLAKY Slang term -Alludes to less than acceptable performance.

FILE AREA The physical location of the file, on the disk, or in memory.

'FPDE' Initials for File Primary Directory Entry; a file's entry and file area pointers in the disk directory.

'FXDE' Initials for File Extended Directory Entry; a file's entry and file area pointers, in the case of an overflow in the 'FPDE'.

GAT Initials for Granule Allocation Table; A table from which available file areas are assigned to file entries.

GRANULE Unit of 5 sectors. On the TRS-80 disk operating system, a 'granule' is the basic unit of disk storage allocation. The diskette 'DIRECTORY' file keeps track of free and assigned disk space in terms of 'granules'.

HASH CODE A code number generated and used as a direct addressing technique in which the key is converted to a pseudo-random number from which the required address is derived.

HEADER RECORD A record containing common, constant or identifying information for a group of records which follow.

HEXADECIMAL See 'BASE 16'

HIT Initials for Hash Index Table; an addressing technique in which a disk file is referenced by a code number in a table, and the position of that code in the table relates to the file entry in the directory.

INDEX A table used to determine the location of a record.

INDIRECT ADDRESSING Any method of specifying or locating a storage location whereby the key (of itself or through calculation) does not represent an address. For example, locating an address through indices.

APPENDIX A

INSTRING (INSTRING SEARCH) Refers to the capability of locating a substring of characters that may exist in another character string. An example would be: Substring = "THE" String = "NOW IS THE TIME". An INSTRING routine would locate the substring and return its starting position within that string. In this example, it would return a value of eight.

INTEGER A natural or whole number. In the TRS-80, integer values may not exceed the range of +32767 to -32768.

INVERTED FILE A file structure which permits fast spontaneous searching for previous unspecified information. Independent lists or indices are maintained in records' keys which are accessible according to the values of specific fields.

INVERTED LIST A list organized by a secondary key --- not a primary key.

IPL Initials for Initial Program Loader; a program usually executed upon pressing of the 'RESET' button.

KEY A data item used to identify or locate a record or other data grouping.

LABEL A set of symbols used to identify or describe an item, record, message or file. Occasionally, it may be the same as the address in storage.

LEAST SIGNIFICANT BYTE The significant byte contributing the smallest quantity to the value of a numeral.

LIST An ordered set of data items. A 'chain'.

LOAD MODULE A program developed for loading into storage and being executed when control is passed to the program.

LOCK-OUT (TRACKS) Unusable tracks, on the disk, that are not accessible because of damage or by user option.

LOGICAL An adjective describing the form of data organization, hardware or system that is perceived by an application program, programmer, or user; it may be different than the real (PHYSICAL) form.

LOGICAL DATA-BASE DESCRIPTION A schema. A description of the overall data-base structure, as perceived for the users, which is employed by the data base management software.

LOGICAL FILE A file as perceived by an application program; it may be in a completely different form from that in which it is stored on the storage units.

APPENDIX A

LOGICAL OPERATOR A mathematical symbol that represents a mathematical process to be performed on an associated operand. Such operators are 'AND', 'OR', 'NOT', 'AND NOT' and 'OR NOT'.

LOGICAL RECORD A record or data item as perceived by an application program; it may be in a completely different form from that in which it is stored on the storage units.

LSB See LEAST SIGNIFICANT BYTE.

MACHINE LANGUAGE Direct machine readable code.

MAINTENANCE OF A FILE (1) The addition, deletion, changing or updating of records in the database. (2) Periodic reorganization of a file to better accommodate items that have been added.

MONITOR A program that may supervise the operation of another program for operation or debugging or other purposes.

MOST SIGNIFICANT BYTE The significant byte contributing the greatest quantity to the value of a numeral.

MSB See MOST SIGNIFICANT BYTE.

MULTIPLE-KEY RETRIEVAL Retrieval which requires searches of data based on the values of several key fields (some or all of which are secondary keys).

NULL An absence of information as contrasted with zero or blank for the presence of no information.

NYBBLE The four right most or left most binary digits of a byte.

ON-LINE An on-line system is one in which the input data enter the computer directly from their point of origin, and/or output data are transmitted directly to where they are used. The intermediate stages such as writing tape, loading disks or off-line printing are avoided.

ON-LINE STORAGE Storage devices and especially the storage media which they contain under the direct control of a computing system, not off-line or in a volume library.

OPEN RECORDS TABLE (ORT) A table of binary values in which the relative position of a single value determines the status of a record with the same relative position; i.e., the Nth binary number determines the status of the Nth record. EXAMPLE: If the 8th binary number in the table is a zero, then the 8th record is open. Conversely, if the 8th binary number in the table is a one, then the 8th record is on file.

OPERATING SYSTEM Software which enables a computer to supervise its own operations, automatically calling in programs, routines, language and data as needed for continuous throughput of different types of jobs.

PARITY Parity relates to the maintenance of a sameness of level or count, i.e., keeping the same number of binary ones in a computer word and thus be able to perform a check based on an even or odd number for all words under examination.

PHYSICAL An adjective, contrasted with logical, which refers to the form in which data or systems exist in reality. Data is often converted by software from the form in which it is physically stored to a form in which a user or programmer perceives it.

PHYSICAL DATA BASE A data base in the form in which it is stored on the storage media, including pointers or other means of interconnecting it. Multiple logical data bases may be derived from one or more physical data bases.

PHYSICAL RECORD A collection of bits that are physically recorded on the storage medium and which are read or written by one machine input/output instruction.

POINTER The address or a record (or other data groupings) contained in another record so that a program may access the former record when it has retrieved the latter record. The address can be absolute, relative or symbolic, hence, the pointer is referred to as absolute, relative or symbolic.

PRIMARY ENTRY The main entry made to the directory. Also see 'FPDE'.

RANDOM ACCESS To obtain data directly from any storage location regardless of its position, with respect to the previously referenced information. Also called 'DIRECT ACCESS'.

RANDOM ACCESS STORAGE A storage technique in which the time required to obtain information is independent of the location of the information most recently obtained.

READ To accept or copy information or data from input devices or a memory register; i.e., to read out, to read in.

RECORD A group of related fields of information treated as a unit by an application program.

RELATIONAL OPERATOR A mathematical symbol that represents a mathematical process to perform a comparison describing the relationship between two values (< less than....> greater than... = equal.... <> not equal... and combinations thereof (see TRS-80 LEVEL II manual, Section 1, Page 5). On the TRS-80, relational comparisons may be made on string values as well as numerical values.

RELATIVE (as pertains to position) An address or position that is referenced to a point of origin; i.e. X+20 is a specific position, 20 places from the reference point. If the reference point was at 50, then the absolute position would be at 70 (50+20=70). Also, 50 (since it is the starting reference point) is at relative position 0.

SCHEMA A map of the overall logical structure of a database.

APPENDIX A

SEARCH To examine a series of items for any that have a desired property or properties.

SECONDARY INDEX An index composed of secondary keys rather than primary keys.

SECTOR The smallest addressable portion of storage on a diskette (a unit of 256 bytes on a TRS-80 diskette).

SEEK To position the access mechanism of a direct-access storage device at a specified location.

SEQUENTIAL ACCESS Access in which records must be read serially or sequentially one after the other; i.e., ASCII files, tape.

SINGLE PRECISION A positive or negative numerical value of 6 digits in length, not including a decimal point (EXAMPLE: 99999.9).

SORT To arrange a file or data in a sequence by a specified key (may be alphabetic or numeric and in descending or ascending order).

SOURCE CODE The text from which code that may be executed is derived.

SYSTEM FILE A program used by the operating system to manage the executing program and/or the computer's resources.

SUB-STRINGS SUB-STRING SEARCH See INSTRING

TABLE A collection of data suitable for quick reference, each item being uniquely identified either by a label or its relative position.

TALLY To add or subtract a digit from a quantity.

TOKEN A one byte code representing a larger word consisting of 2 or more characters.

TRACK The circular recording surface traversed by a read/write head on the disk. On the TRS-80 a track contains 10 sectors (2 granules).

TRANSACTION An input record applied to an established file. The input record describes some "event" that will either cause a new file record to be generated, an existing record to be changed or an existing record to be deleted.

TRANSPARENT Complexities that are hidden from the programmers or users (made transparent to them) by the software.

VECTOR A line representing the properties of magnitude and direction. Since such a 'line' can be described in mathematical terms, a mathematical description (expressed in numbers, of course) of a given 'direction' and 'magnitude' is referred to as a "vector".

APPENDIX A

VERIFY To check a data transfer or transcription.

WORKING STORAGE A portion of storage, usually computer main memory, reserved for the temporary results of operations.

WRITE To record information on a storage device.

ZAP To change a byte or bytes of data in memory or on diskette by using a software utility program.

ZEROETH Zeroeth is to '0' as first is to '1'; in computer terms the first position of anything is usually described as the 'zeroeth' and the next position is the 'first' and so on.



APPENDIX A

II LEVEL II 'BASIC TOKENS'

Program statements, in LEVEL II and DISK BASIC are not stored in memory as they are typed and viewed on the video display. For instance 'PRINT' is stored as the single byte character: "?". The following is a list of LEVEL II TOKENS in the following format:

HEX-DECIMAL 'BASIC' KEYWORD

80-128	END	AA-170	KILL	D4-212	>
81-129	FOR	AB-171	LSET	D5-213	=
82-130	RESET	AC-172	RSET	D6-214	<
83-131	SET	AD-173	SAVE	D7-215	SGN
84-132	CLS	AE-174	SYSTEM	D8-216	INT
85-133	CMD	AF-175	LPRINT	D9-217	ABS
86-134	RANDOM	B0-176	DEF	DA-218	FRE
87-135	NEXT	B1-177	POKE	DB-219	INP
88-136	DATA	B2-178	PRINT	DC-220	POS
89-137	INPUT	B3-179	CONT	DD-221	SQR
8A-138	DIM	B4-180	LIST	DE-222	RND
8B-139	READ	B5-181	LLIST	DF-223	LOG
8C-140	LET	B6-182	DELETE	E0-224	EXP
8D-141	GOTO	B7-183	AUTO	E1-225	COS
8E-142	RUN	B8-184	CLEAR	E2-226	SIN
8F-143	IF	B9-185	CLOAD	E3-227	TAN
90-144	RESTORE	BA-186	CSAVE	E4-228	ATN
91-145	GOSUB	BB-187	NEW	E5-229	PEEK
92-146	RETURN	BC-188	TAB	E6-230	CVI
93-147	REM	BD-189	TO	E7-231	CVS
94-148	STOP	BE-190	FN	E8-232	CVD
95-149	ELSE	BF-191	USING	E9-233	EOF
96-150	TRON	C0-192	VARPTR	EA-234	LOC
97-151	TROFF	C1-193	USR	EB-235	LOF
98-152	DEFSTR	C2-194	ERL	EC-236	MKIS
99-153	DEFINT	C3-195	ERR	ED-237	MKS\$
9A-154	DEFSNG	C4-196	STRING\$	EE-238	MKD\$
9B-155	DEFDBL	C5-197	INSTR	EF-239	CINT
9C-156	LINE	C6-198	POINT	F0-240	CSNG
9D-157	EDIT	C7-199	TIME\$	F1-241	CDBL
9E-158	ERROR	C8-200	MEM	F2-242	FIX
9F-159	RESUME	C9-201	INKEY\$	F3-243	LEN
A0-160	OUT	CA-202	THEN	F4-244	STR\$
A1-161	ON	CB-203	NOT	F5-245	VAL
A2-162	OPEN	CC-204	STEP	F6-246	ASC
A3-163	FIELD	CD-205	+	F7-247	CHR\$
A4-164	GET	CE-206	-	F8-248	LEFT\$
A5-165	PUT	CF-207	*	F9-249	RIGHT\$
A6-166	CLOSE	F0-208	/	FA-250	MID\$
A7-167	LOAD	D1-209	(UP ARROW)	FB-251	**
A8-168	MERGE	D2-210	AND	FC-252	**
A9-169	NAME **	D3-211	OR	FD-253	**
				FE-254	**
				FF-255	ISA **

** = NOT USED BY SYSTEM

APPENDIX A TRS DOS 2.2 DIRECTORY

(GAT SECTOR) (figure A1.1)

```

311000  FFFF FFFF FFFC FCFC FCFC FCFC FEFF FFFF .....
311010  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311020  FCFC FCFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311030  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311040  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311050  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311060  FCFC FCFC FCFC FCFC FCFC FCFC FCFC FCFC .....
311070  FCFC FCFC FCFC FCFC FCFC FCFC FCFC FCFC .....
311080  FCFC FCFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311090  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110A0  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110B0  FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110C0  FFFF FFFF FFFF FFFF FFFF FF21 0000 E042 .....!...B
3110D0  5452 5344 4F53 2020 3035 2F32 312F 3739 TRSDOS..05/21/79
3110E0  0D0D FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110F06 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....

```

Relative sector 0, track 11
 35 track TRSDOS 2.2
 Master disk password 'HASH' code = E042

(HIT SECTOR) (figure A1.2)

```

311100  A22C 2E2F 2C2D 2A2B 0000 0000 0000 0000 .../,-*+.....
311110  0000 0000 0000 0000 0000 0000 0000 0000 .....
311120  2800 0000 0000 0000 0000 0000 0000 0000 (.....
311130  0000 0000 0000 0000 0000 0000 0000 0000 .....
311140  F2C5 0074 006C 0000 0000 0000 0000 0000 .....
311150  0000 0000 0000 0000 0000 0000 0000 0000 .....
311160  0000 00E3 F069 0000 0000 0000 0000 0000 .....
311170  0000 0000 0000 0000 0000 0000 0000 0000 .....
311180  0000 0000 0000 0000 0000 0000 0000 0000 .....
311190  0000 0000 0000 0000 0000 0000 0000 0000 .....
3111A0  0000 0000 0080 0000 0000 0000 0000 0000 .....
3111B0  0000 0000 0000 0000 0000 0000 0000 0000 .....
3111C0  0000 0000 007C 4B00 0000 0000 0000 0000 .....K.....
3111D0  0000 0000 0000 0000 0000 0000 0000 0000 .....
3111E0  0000 0000 0000 0000 0000 0000 0000 0000 .....
3111F06 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

Relative sector 1, track 11

A2 = BOOT/SYS	2F = SYS1/SYS	6C = BASICR/CMD
28 = SYS6/SYS	74 = TEST1/CMD	69 = GETDISK/BAS
F2 = FORMAT/CMD	E3 = TEST2/BAS	80 = DISKDUMP/BAS
2C = DIR/SYS	2C = SYS2/SYS	7C = GETTAPE/BAS
C5 = BACKUP/CMD	F0 = BASIC/CMD	2A = SYS4/SYS
2E = SYS0/SYS	2D = SYS3/SYS	4B = TAPEDISK/CMD
		2B = SYS5/SYS

(FPDE/FXDE SECTOR 1) (figure A1.3)

311200	5E00	0000	0042	4F4F	5420	2020	2053	5953BOOT.....SYS
311210	607F	1FB2	0500	0000	FFFF	0000	0000	0000
311220	5F00	0000	0053	5953	3620	2020	2053	5953SYS6.....SYS
311230	EB29	210E	0F00	1322	FFFF	FFFF	FFFF	FFFF	.) !....."
311240	1E00	0000	0046	4F52	4D41	5420	2043	4D44FORMAT..CMD
311250	982F	9642	0F00	0202	FFFF	FFFF	FFFF	FFFF	./..B.....
311260	0000	0000	0000	0000	0000	0000	0000	0000
311270	0000	0000	0000	0000	0000	0000	0000	0000
311280	0000	0000	0000	0000	0000	0000	0000	0000
311290	0000	0000	0000	0000	0000	0000	0000	0000
3112A0	0000	0000	0000	0000	0000	0000	0000	0000
3112B0	0000	0000	0000	0000	0000	0000	0000	0000
3112C0	0000	0000	0000	0000	0000	0000	0000	0000
3112D0	0000	0000	0000	0000	0000	0000	0000	0000
3112E0	0000	0000	0000	0000	0000	0000	0000	0000
3112F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 2

BOOT/SYS = TRACK 00, SECTOR 0
 SYS6/SYS = TRACK 13, SECTOR 5
 FORMAT/CMD = TRACK 02, SECTOR 0

(FPDE/FXDE SECTOR 2) (figure A1.4)

311300	5D00	0000	0044	4952	2020	2020	2053	5953DIR.....SYS
311310	A71D	F9E5	0A00	1101	FFFF	0000	0000	0000
311320	0000	0000	0000	0000	0000	0000	0000	0000
311330	0000	0000	0000	0000	0000	0000	0000	0000
311340	1E00	0000	0042	4143	4B55	5020	2043	4D44BACKUP..CMD
311350	ACA8	9642	0F00	0322	FFFF	FFFF	FFFF	FFFF	...B..."
311360	0000	0000	0000	0000	0000	0000	0000	0000
311370	0000	0000	0000	0000	0000	0000	0000	0000
311380	0000	0000	0000	0000	0000	0000	0000	0000
311390	0000	0000	0000	0000	0000	0000	0000	0000
3113A0	0000	0000	0000	0000	0000	0000	0000	0000
3113B0	0000	0000	0000	0000	0000	0000	0000	0000
3113C0	0000	0000	0000	0000	0000	0000	0000	0000
3113D0	0000	0000	0000	0000	0000	0000	0000	0000
3113E0	0000	0000	0000	0000	0000	0000	0000	0000
3113F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 3

DIR/SYS = TRACK 11, SECTOR 0
 BACKUP/CMD = TRACK 03, SECTOR 5

APPENDIX A TRS DOS 2.2 DIRECTORY

(FPDE/FXDE SECTOR 3) (figure A1.5)

311400	5F00	0000	0053	5953	3020	2020	2053	5953SYS0.....SYS
311410	EB29	210E	0F00	0022	FFFF	FFFF	FFFF	FFFF	.)!....."
311420	0000	0000	0000	0000	0000	0000	0000	0000
311430	0000	0000	0000	0000	0000	0000	0000	0000
311440	0000	0000	0000	0000	0000	0000	0000	0000
311450	0000	0000	0000	0000	0000	0000	0000	0000
311460	0000	0000	0000	0000	0000	0000	0000	0000
311470	0000	0000	0000	0000	0000	0000	0000	0000
311480	0000	0000	0000	0000	0000	0000	0000	0000
311490	0000	0000	0000	0000	0000	0000	0000	0000
3114A0	0000	0000	0000	0000	0000	0000	0000	0000
3114B0	0000	0000	0000	0000	0000	0000	0000	0000
3114C0	0000	0000	0000	0000	0000	0000	0000	0000
3114D0	0000	0000	0000	0000	0000	0000	0000	0000
3114E0	0000	0000	0000	0000	0000	0000	0000	0000
3114F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 4

SYS0/SYS = TRACK 00, SECTOR 5

(FPDE/FXDE SECTOR 4) (figure A1.6)

311500	5F00	0000	0053	5953	3120	2020	2053	5953SYS1.....SYS
311510	EB29	210E	0500	1000	FFFF	FFFF	FFFF	FFFF	.)!.....
311520	0000	0000	0000	0000	0000	0000	0000	0000
311530	0000	0000	0000	0000	0000	0000	0000	0000
311540	1000	0000	0054	4553	5431	2020	2043	4D44TEST1...CMD
311550	9642	9642	0600	1501	FFFF	FFFF	FFFF	FFFF	.B.B.....
311560	1000	0097	0054	4553	5432	2020	2042	4153TEST2...BAS
311570	9642	9642	3900	0C26	1603	1D00	FFFF	FFFF	.B.B9...&.....
311580	0000	0000	0000	0000	0000	0000	0000	0000
311590	0000	0000	0000	0000	0000	0000	0000	0000
3115A0	0000	0000	0000	0000	0000	0000	0000	0000
3115B0	0000	0000	0000	0000	0000	0000	0000	0000
3115C0	0000	0000	0000	0000	0000	0000	0000	0000
3115D0	0000	0000	0000	0000	0000	0000	0000	0000
3115E0	0000	0000	0000	0000	0000	0000	0000	0000
3115F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 5

SYS1/SYS = TRACK 10, SECTOR 0
 TEST1/CMD = TRACK 15, SECTOR 0
 TEST2/BAS = TRACK 0C, SECTOR 5

(FPDE/FXDE SECTOR 5)

(figure A1.7)

311600	5F00	0000	0053	5953	3220	2020	2053	5953SYS2.....SYS
311610	EB29	210E	0500	1020	FFFF	FFFF	FFFF	FFFF	.)!.....
311620	0000	0000	0000	0000	0000	0000	0000	0000
311630	0000	0000	0000	0000	0000	0000	0000	0000
311640	0000	0000	0000	0000	0000	0000	0000	0000
311650	0000	0000	0000	0000	0000	0000	0000	0000
311660	1E00	0000	0042	4153	4943	2020	2043	4D44BASIC...CMD
311670	782F	9642	1400	1903	FFFF	FFFF	FFFF	FFFF	./.B.....
311680	0000	0000	0000	0000	0000	0000	0000	0000
311690	0000	0000	0000	0000	0000	0000	0000	0000
3116A0	0000	0000	0000	0000	0000	0000	0000	0000
3116B0	0000	0000	0000	0000	0000	0000	0000	0000
3116C0	0000	0000	0000	0000	0000	0000	0000	0000
3116D0	0000	0000	0000	0000	0000	0000	0000	0000
3116E0	0000	0000	0000	0000	0000	0000	0000	0000
3116F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 6

SYS2/SYS = TRACK 10, SECTOR 5
 BASIC/CMD = TRACK 19, SECTOR 0

(FPDE/FXDE SECTOR 6)

(figure A1.8)

311700	5F00	0000	0053	5953	3320	2020	2053	5953SYS3.....SYS
311710	EB29	210E	0500	1200	FFFF	FFFF	FFFF	FFFF	.)!.....
311720	0000	0000	0000	0000	0000	0000	0000	0000
311730	0000	0000	0000	0000	0000	0000	0000	0000
311740	1E00	0000	0042	4153	4943	5220	2043	4D44BASICR..CMD
311750	782F	9642	1700	1D24	FFFF	FFFF	FFFF	FFFF	./.B...\$......
311760	1000	0005	0047	4554	4449	534B	2042	4153GETDISK.BAS
311770	9642	9642	0700	1B01	FFFF	FFFF	FFFF	FFFF	.B.B.....
311780	0000	0000	0000	0000	0000	0000	0000	0000
311790	0000	0000	0000	0000	0000	0000	0000	0000
3117A0	1000	00CF	0044	4953	4B44	554D	5042	4153DISKDUMPBAS
3117B0	9642	9642	0300	1820	FFFF	FFFF	FFFF	FFFF	.B.B.....
3117C0	1000	00AE	0047	4554	5441	5045	2042	4153GETTAPE.BAS
3117D0	9642	9642	0500	1C01	FFFF	FFFF	FFFF	FFFF	.B.B.....
3117E0	0000	0000	0000	0000	0000	0000	0000	0000
3117F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 7

SYS3/SYS = TRACK 12, SECTOR 0
 BASICR/CMD = TRACK 10, SECTOR 5
 GETDISK/BAS = TRACK 1B, SECTOR 0
 DISKDUMP/BAS = TRACK 18, SECTOR 5
 GETTAPE/BAS = TRACK 1C, SECTOR 0

APPENDIX A TRS DOS 2.2 DIRECTORY

————(FPDE/FXDE SECTOR 7)————(figure A1.9)————

311800	5F00	0000	0053	5953	3420	2020	2053	5953SYS4.....SYS
311810	EB29	210E	0500	1220	FFFF	FFFF	FFFF	FFFF	.)!.....
311820	0000	0000	0000	0000	0000	0000	0000	0000
311830	0000	0000	0000	0000	0000	0000	0000	0000
311840	0000	0000	0000	0000	0000	0000	0000	0000
311850	0000	0000	0000	0000	0000	0000	0000	0000
311860	0000	0000	0000	0000	0000	0000	0000	0000
311870	0000	0000	0000	0000	0000	0000	0000	0000
311880	0000	0000	0000	0000	0000	0000	0000	0000
311890	0000	0000	0000	0000	0000	0000	0000	0000
3118A0	0000	0000	0000	0000	0000	0000	0000	0000
3118B0	0000	0000	0000	0000	0000	0000	0000	0000
3118C0	1000	0000	0054	4150	4544	4953	4B43	4D44TAPEDISKCMD
3118D0	9642	9642	0200	1800	FFFF	FFFF	FFFF	FFFF	.B.B.....
3118E0	0000	0000	0000	0000	0000	0000	0000	0000
3118F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 8

SYS4/SYS = TRACK 12, SECTOR 5
TAPEDISK/CMD = TRACK 18, SECTOR 0

————(FPDE/FXDE SECTOR 8)————(figure A1.10)————

311900	5F00	0000	0053	5953	3520	2020	2053	5953SYS5.....SYS
311910	EB29	210E	0500	1300	FFFF	FFFF	FFFF	FFFF	.)!.....
311920	0000	0000	0000	0000	0000	0000	0000	0000
311930	0000	0000	0000	0000	0000	0000	0000	0000
311940	0000	0000	0000	0000	0000	0000	0000	0000
311950	0000	0000	0000	0000	0000	0000	0000	0000
311960	0000	0000	0000	0000	0000	0000	0000	0000
311970	0000	0000	0000	0000	0000	0000	0000	0000
311980	0000	0000	0000	0000	0000	0000	0000	0000
311990	0000	0000	0000	0000	0000	0000	0000	0000
3119A0	0000	0000	0000	0000	0000	0000	0000	0000
3119B0	0000	0000	0000	0000	0000	0000	0000	0000
3119C0	0000	0000	0000	0000	0000	0000	0000	0000
3119D0	0000	0000	0000	0000	0000	0000	0000	0000
3119E0	0000	0000	0000	0000	0000	0000	0000	0000
3119F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 9

SYS5/SYS = TRACK 13, SECTOR 0

APPENDIX A NEW DOS 2.1 DIRECTORY

(GAT SECTOR) (figure A2.1)

```

311000 FFFF FFFF FFFF FFFF FFFF FFFF FEFD FEFD .....
311010 FFFF FFFF FFFF FFFF FDFF FFFF FFFF FFFF .....
311020 FFFF FFFC FCFC FCFC FFFF FFFF FFFF FFFF .....
311030 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311040 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311050 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
311060 FCFC FCFC FCFC FCFC FCFC FCFC FCFC FCFC .....
311070 FCFC FCFC FCFC FCFC FCFC FCFC FCFC FCFC .....
311080 FCFC FCFC FCFC FCFC FFFF FFFF FFFF FFFF .....
311090 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110A0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110B0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .....
3110C0 FFFF FFFF FFFF FFFF FFFF EF21 0000 E042 .....!...B
3110D0 4E45 5744 4F53 3430 3034 2F33 312F 3739 NEWDOS4004/31/79
3110E0 0D0D 2020 2020 2020 2020 2020 2020 2020 .....
3110F06 2020 2020 2020 2020 2020 2020 2020 2020 .....

```

Relative sector 0, track 11

40 track NEW DOS+

Master disk password 'HASH' code = E042

(HIT SECTOR) (figure A2.2)

```

311100 A22C 2E2F 2C2D 2A2B 0000 0000 0000 0000 ..,./,-*+.....
311110 0000 0000 0000 0000 0000 0000 0000 0000 .....
311120 2800 0000 00A7 26A6 0000 0000 0000 0000 (. ....&.....
311130 0000 0000 0000 0000 0000 0000 0000 0000 .....
311140 F200 0000 0000 0000 0000 0000 0000 0000 .....
311150 0000 0000 0000 0000 0000 0000 0000 0000 .....
311160 0000 0000 0500 0000 0000 0000 0000 0000 .....
311170 0000 0000 0000 0000 0000 0000 0000 0000 .....
311180 8055 00EE 0000 0000 0000 0000 0000 0000 .U.....
311190 0000 0000 0000 0000 0000 0000 0000 0000 .....
3111A0 F000 0000 4632 0089 0000 0000 0000 0000 ....F2.....
3111B0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3111C0 6F67 0000 0000 0000 0000 0000 0000 0000 .....
3111D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3111E0 0000 0079 0000 0000 0000 0000 0000 0000 .....
3111F06 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

Relative sector 1, track 11

A2 = BOOT/SYS

28 = SYS6/SYS

F2 = FORMAT/CMD

80 = DISKDUMP/CMD

6F = BASIC/CMD

2C = DIR/SYS

55 = LEVEL1/CMD

67 = COPY/CMD

2E = SYS0/SYS

2F = SYS1/SYS

EE = DIRCHECK/CMD

79 = SUPERZAP

2C = SYS2/SYS

05 = SUPERZAP/COM

46 = DISASSEM/CMD

89 = EDTASH/CMD

2D = SYS3/SYS

A7 = SYS11/SYS

32 = LMOFFSET/CMD

2A = SYS4/SYS

26 = SYS12/SYS

2B = SYS5/SYS

A6 = SYS13/SYS

(FPDE/FXDE SECTOR 1)

(figure A2.3)

311200	5E00	0000	0042	4F4F	5420	2020	2053	5953ROOT....SYS
311210	607F	1FB2	0500	0000	FFFF	0000	0000	0000
311220	5F00	0021	0053	5953	3620	2020	2053	5953	...!.SYS6....SYS
311230	EB29	210E	0E00	1322	FFFF	FFFF	FFFF	FFFF	.)!....".....
311240	1E00	0008	0046	4F52	4D41	5420	2043	4D44FORMAT..CMD
311250	8130	9642	0F00	0202	FFFF	FFFF	FFFF	FFFF	.0.B.....
311260	0000	0000	0000	0000	0000	0000	0000	0000
311270	0000	0000	0000	0000	0000	0000	0000	0000
311280	1000	00F2	0044	4953	4B44	554D	5042	4153DISKDUMPBAS
311290	9642	9642	0A00	0B01	FFFF	FFFF	FFFF	FFFF	.B.B.....
3112A0	1E00	0077	0042	4153	4943	2020	2043	4D44BASIC...CMD
3112B0	8130	9642	1400	0322	0920	FFFF	FFFF	FFFF	.0.B...".....
3112C0	1000	008D	004C	5631	4453	4B53	4C43	4D44LV1DSKSLCMD
3112D0	9642	9642	0300	0F00	FFFF	FFFF	FFFF	FFFF	.B.B.....
3112E0	0000	0000	0000	0000	0000	0000	0000	0000
3112F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 2

BOOT/SYS = TRACK 00, SECTOR 0
 SYS6/SYS = TRACK 13, SECTOR 5
 FORMAT/CMD = TRACK 02, SECTOR 0
 DISKDUMP/BAS = TRACK 0B, SECTOR 0
 BASIC/CMD = TRACK 03, SECTOR 5 (EXTENT 1)
 TRACK 09, SECTOR 5 (EXTENT 2)

(FPDE/FXDE SECTOR 2)

(figure A2.4)

311300	5D00	0000	0044	4952	2020	2020	2053	5953DIR.....SYS
311310	A71D	F9E5	0A00	1101	FFFF	0000	0000	0000
311320	0000	0000	0000	0000	0000	0000	0000	0000
311330	0000	0000	0000	0000	0000	0000	0000	0000
311340	0000	0000	0000	0000	0000	0000	0000	0000
311350	0000	0000	0000	0000	0000	0000	0000	0000
311360	0000	0000	0000	0000	0000	0000	0000	0000
311370	0000	0000	0000	0000	0000	0000	0000	0000
311380	1000	0000	004C	4556	454C	3120	2043	4D44LEVEL1..CMD
311390	9642	9642	1300	0521	0900	0A00	FFFF	FFFF	.B.B...!.....
3113A0	0000	0000	0000	0000	0000	0000	0000	0000
3113B0	0000	0000	0000	0000	0000	0000	0000	0000
3113C0	1E00	00FD	0043	4F50	5920	2020	2043	4D44COPY....CMD
3113D0	8130	9642	0500	0620	FFFF	FFFF	FFFF	FFFF	.0.B.....
3113E0	0000	0000	0000	0000	0000	0000	0000	0000
3113F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 3

DIR/SYS = TRACK 11, SECTOR 0
 LEVEL1/CMD = TRACK 05, SECTOR 5 (EXTENT 1)
 TRACK 09, SECTOR 0 (EXTENT 2)
 TRACK 0A, SECTOR 0 (EXTENT 3)
 COPY/CMD = TRACK 06, SECTOR 5

(FPDE/FXDE SECTOR 3) (figure A2.5)

```

311400 5F00 005D 0053 5953 3020 2020 2053 5953 .....SYS0.....SYS
311410 EB29 210E 0D00 0022 FFFF FFFF FFFF FFFF .)!.....".....
311420 0000 0000 0000 0000 0000 0000 0000 0000 .....
311430 0000 0000 0000 0000 0000 0000 0000 0000 .....
311440 0000 0000 0000 0000 0000 0000 0000 0000 .....
311450 0000 0000 0000 0000 0000 0000 0000 0000 .....
311460 0000 0000 0000 0000 0000 0000 0000 0000 .....
311470 0000 0000 0000 0000 0000 0000 0000 0000 .....
311480 0000 0000 0000 0000 0000 0000 0000 0000 .....
311490 0000 0000 0000 0000 0000 0000 0000 0000 .....
3114A0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3114B0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3114C0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3114D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3114E0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3114F06 0000 0000 0000 0000 0000 0000 0000 0000 .....

```

Relative sector 4

SYS0/SYS = TRACK 00, SECTOR 5

(FPDE/FXDE SECTOR 4) (figure A2.6)

```

311500 5F00 008E 0053 5953 3120 2020 2053 5953 .....SYS1.....SYS
311510 EB29 210E 0500 1000 FFFF FFFF FFFF FFFF .)!.....
311520 0000 0000 0000 0000 0000 0000 0000 0000 .....
311530 0000 0000 0000 0000 0000 0000 0000 0000 .....
311540 0000 0000 0000 0000 0000 0000 0000 0000 .....
311550 0000 0000 0000 0000 0000 0000 0000 0000 .....
311560 0000 0000 0000 0000 0000 0000 0000 0000 .....
311570 0000 0000 0000 0000 0000 0000 0000 0000 .....
311580 1000 00EC 0044 4952 4348 4543 4B43 4D44 .....DIRCHECKCMD
311590 9642 9642 0D00 0D00 0E20 0500 FFFF FFFF .B.B.....
3115A0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3115B0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3115C0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3115D0 0000 0000 0000 0000 0000 0000 0000 0000 .....
3115E0 1000 006A 0053 5550 4552 5A41 5020 2020 .....SUPERZAP...
3115F06 9642 9642 3600 1905 1F20 2023 FFFF FFFF .B.B6.....#.....

```

Relative sector 5

```

SYS1/SYS      = TRACK 10, SECTOR 0
DIRCHECK/CMD = TRACK 0D, SECTOR 0 (EXTENT 1)
               TRACK 0E, SECTOR 5 (EXTENT 2)
               TRACK 05, SECTOR 0 (EXTENT 3)
SUPERZAP      = TRACK 19, SECTOR 0 (EXTENT 1)
               TRACK 1F, SECTOR 5 (EXTENT 2)
               TRACK 20, SECTOR 5 (EXTENT 3)

```


APPENDIX A NEW DOS 2.1 DIRECTORY

(FPDE/FXDE SECTOR 5) (figure A2.7)

311600	5F00	0034	0053	5953	3220	2020	2053	5953	...4.SYS2....SYS
311610	EB29	210E	0500	1020	FFFF	FFFF	FFFF	FFFF	.)!.....
311620	0000	0000	0000	0000	0000	0000	0000	0000
311630	0000	0000	0000	0000	0000	0000	0000	0000
311640	0000	0000	0000	0000	0000	0000	0000	0000
311650	0000	0000	0000	0000	0000	0000	0000	0000
311660	1000	00BD	0053	5550	4552	5A41	5043	4F4DSUPERZAPCOM
311670	9642	9642	1A00	1523	0A20	1500	FFFF	FFFF	.B.B...#.....
311680	0000	0000	0000	0000	0000	0000	0000	0000
311690	0000	0000	0000	0000	0000	0000	0000	0000
3116A0	1000	0086	0044	4953	4153	5345	4D43	4D44DISASSEMBCMD
3116B0	9642	9642	1400	0703	FFFF	FFFF	FFFF	FFFF	.B.B.....
3116C0	0000	0000	0000	0000	0000	0000	0000	0000
3116D0	0000	0000	0000	0000	0000	0000	0000	0000
3116E0	0000	0000	0000	0000	0000	0000	0000	0000
3116F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 6

SYS2/SYS = TRACK 10, SECTOR 5
 SUPERZAP/COM = TRACK 15, SECTOR 5 (EXTENT 1)
 TRACK 0A, SECTOR 5 (EXTENT 2)
 TRACK 15, SECTOR 0 (EXTENT 3)
 DISASSEMB/CMD = TRACK 07, SECTOR 0

(FPDE/FXDE SECTOR 6) (figure A2.8)

311700	5F00	004C	0053	5953	3320	2020	2053	5953	...L.SYS3....SYS
311710	EB29	210E	0500	1200	FFFF	FFFF	FFFF	FFFF	.)!.....
311720	5F00	00EC	0053	5953	3131	2020	2053	5953SYS11...SYS
311730	EB29	210E	0500	2000	FFFF	FFFF	FFFF	FFFF	.)!.....
311740	0000	0000	0000	0000	0000	0000	0000	0000
311750	0000	0000	0000	0000	0000	0000	0000	0000
311760	0000	0000	0000	0000	0000	0000	0000	0000
311770	0000	0000	0000	0000	0000	0000	0000	0000
311780	0000	0000	0000	0000	0000	0000	0000	0000
311790	0000	0000	0000	0000	0000	0000	0000	0000
3117A0	1000	00E0	004C	4D4F	4646	5345	5443	4D44LMOFFSETCMD
3117B0	9642	9642	0700	1721	FFFF	FFFF	FFFF	FFFF	.B.B...!.....
3117C0	0000	0000	0000	0000	0000	0000	0000	0000
3117D0	0000	0000	0000	0000	0000	0000	0000	0000
3117E0	0000	0000	0000	0000	0000	0000	0000	0000
3117F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 7

SYS3/SYS = TRACK 12, SECTOR 0
 SYS11/SYS = TRACK 20, SECTOR 0
 LMOFFSET/CMD = TRACK 17, SECTOR 5

APPENDIX A NEW DOS 2.1 DIRECTORY

(FPDE/FXDE SECTOR 7) (figure A2.9)

311800	5F00	00BA	0053	5953	3420	2020	2053	5953SYS4.....SYS
311810	EB29	210E	0500	1220	FFFF	FFFF	FFFF	FFFF	.)!.....
311820	5F00	00A4	0053	5953	3132	2020	2053	5953SYS12...SYS
311830	EB29	210E	0500	2220	FFFF	FFFF	FFFF	FFFF	.)!...".....
311840	0000	0000	0000	0000	0000	0000	0000	0000
311850	0000	0000	0000	0000	0000	0000	0000	0000
311860	0000	0000	0000	0000	0000	0000	0000	0000
311870	0000	0000	0000	0000	0000	0000	0000	0000
311880	0000	0000	0000	0000	0000	0000	0000	0000
311890	0000	0000	0000	0000	0000	0000	0000	0000
3118A0	0000	0000	0000	0000	0000	0000	0000	0000
3118B0	0000	0000	0000	0000	0000	0000	0000	0000
3118C0	0000	0000	0000	0000	0000	0000	0000	0000
3118D0	0000	0000	0000	0000	0000	0000	0000	0000
3118E0	0000	0000	0000	0000	0000	0000	0000	0000
3118F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 8

SYS4/SYS = TRACK 12, SECTOR 5
SYS12/SYS = TRACK 22, SECTOR 5

(FPDE/FXDE SECTOR 8) (figure A2.10)

311900	5F00	00CB	0053	5953	3520	2020	2053	5953SYS5.....SYS
311910	EB29	210E	0500	1300	FFFF	FFFF	FFFF	FFFF	.)!.....
311920	5F00	0009	0053	5953	3133	2020	2053	5953SYS13...SYS
311930	EB29	210E	0400	0C20	FFFF	FFFF	FFFF	FFFF	.)!.....
311940	0000	0000	0000	0000	0000	0000	0000	0000
311950	0000	0000	0000	0000	0000	0000	0000	0000
311960	0000	0000	0000	0000	0000	0000	0000	0000
311970	0000	0000	0000	0000	0000	0000	0000	0000
311980	0000	0000	0000	0000	0000	0000	0000	0000
311990	0000	0000	0000	0000	0000	0000	0000	0000
3119A0	1000	009A	0045	4454	4153	4D20	2043	4D44EDTASM..CMD
3119B0	9642	9642	2000	1C06	FFFF	FFFF	FFFF	FFFF	.B.B.....
3119C0	0000	0000	0000	0000	0000	0000	0000	0000
3119D0	0000	0000	0000	0000	0000	0000	0000	0000
3119E0	0000	0000	0000	0000	0000	0000	0000	0000
3119F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 9

SYS5/SYS = TRACK 13, SECTOR 0
SYS13/SYS = TRACK 0C, SECTOR 5
EDTASM/CMD = TRACK 1C, SECTOR 0

APPENDIX A VTOS 3.0 DIRECTORY

(GAT SECTOR) (figure A3.1)

311000	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311010	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311020	FFFF	FDFD	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311030	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311040	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311050	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311060	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC
311070	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC	FCFC
311080	FCFC	FCFC	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
311090	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
3110A0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
3110B0	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF	FFFF
3110C0	FFFF	FFFF	FFFF	FFFF	FFFF	FF30	0000	E042	0...B
3110D0	5654	4F53	3A33	2E30	3038	2F30	322F	3739	VTOS:3.008/02/79	
3110E0	4348	4149	4E20	494E	4954	0D20	2020	2020	CHAIN.INIT.....	
3110F06	2020	2020	2020	2020	2020	2020	2020	2020	2020

Relative sector 0, track 11

35 track VTOS 3.0

Master disk password 'HASH' code = E042

(HIT SECTOR) (figure A3.2)

311100	A2C4	2E2F	2C2D	2A2B	0000	0000	0000	0000	0000	.../,-*+.....
311110	0000	0000	0000	0000	0000	0000	0000	0000	0000
311120	2829	2627	0000	0000	0000	0000	0000	0000	0000	()&'.....
311130	0000	0000	0000	0000	0000	0000	0000	0000	0000
311140	F2C5	E105	6A40	6C2A	0000	0000	0000	0000	0000@.*.....
311150	0000	0000	0000	0000	0000	0000	0000	0000	0000
311160	7ED0	00F3	BDCE	5E9D	0000	0000	0000	0000	0000
311170	0000	0000	0000	0000	0000	0000	0000	0000	0000
311180	0000	00CB	0000	0000	0000	0000	0000	0000	0000
311190	0000	0000	0000	0000	0000	0000	0000	0000	0000
3111A0	00A1	0000	0000	0024	0000	0000	0000	0000	0000\$......
3111B0	0000	0000	0000	0000	0000	0000	0000	0000	0000
3111C0	DA00	0000	0000	0000	0000	0000	0000	0000	0000
3111D0	0000	0000	0000	0000	0000	0000	0000	0000	0000
3111E0	0000	0000	0000	6C00	0000	0000	0000	0000	0000
3111F06	0000	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 1, track 11

A2 = BOOT/SYS	2E = SYS0/SYS	2D = SYS3/SYS
28 = SYS6/SYS	26 = SYS8/SYS	40 = RS232/DVR
F2 = FORMAT/CMD	E1 = PATCH/CMD	CE = BASIC/DOC
7E = VTCOMM/CMD	2F = SYS1/SYS	2A = SYS4/SYS
DA = BASIC/KSM	27 = SYS9/SYS	6C = KSM/DVR
C4 = DIR/SYS	05 = DOLC/DVR	5E = PENCIL/FIX
29 = SYS7/SYS	F3 = VTOS/EPT	6C = INIT/JCL
C5 = BACKUP/CMD	CB = FEATURES/DOC	2B = SYS5/SYS
D0 = COMMAND/DOC	2C = SYS2/SYS	2A = KSR/CMD
A1 = NEWUSER/KSM	6A = PR/DVR	9D = GENERAL/DOC
	BD = VTOS/KSM	24 = UTILITY/DOC

APPENDIX A VTCS 3.0 DIRECTORY

(FPDE/FXDE SECTOR 1) (figure A3.3)

311200	5E00	0000	0042	4F4F	5420	2020	2053	5953BOOT.....SYS
311210	607F	1FB2	0500	0000	FFFF	0000	0000	0000
311220	5F00	0000	0053	5953	3620	2020	2053	5953SYS6.....SYS
311230	EB29	210E	1E00	1325	FFFF	FFFF	FFFF	FFFF	.)!.....%
311240	1E00	0000	0046	4F52	4D41	5420	2043	4D44FORMAT..CMD
311250	2A5F	9642	0F00	0202	FFFF	FFFF	FFFF	FFFF	*..B.....
311260	1E00	0000	0056	5443	4F4D	4D20	2043	4D44VTCOMM..CMD
311270	2A5F	9642	0A00	0801	FFFF	FFFF	FFFF	FFFF	*..B.....
311280	0000	0000	0000	0000	0000	0000	0000	0000
311290	0000	0000	0000	0000	0000	0000	0000	0000
3112A0	0000	0000	0000	0000	0000	0000	0000	0000
3112B0	0000	0000	0000	0000	0000	0000	0000	0000
3112C0	1000	007A	0042	4153	4943	2020	204B	534DBASIC...KSM
3112D0	9642	9642	0100	0900	FFFF	FFFF	FFFF	FFFF	.B.B.....
3112E0	0000	0000	0000	0000	0000	0000	0000	0000
3112F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 2

BOOT/SYS	=	TRACK 00, SECTOR 0
SYS6/SYS	=	TRACK 13, SECTOR 5
FORMAT/CMD	=	TRACK 02, SECTOR 0
VTCOMM/CMD	=	TRACK 08, SECTOR 0
BASIC/KSM	=	TRACK 09, SECTOR 0

(FPDE/FXDE SECTOR 2) (figure A3.4)

311300	5D00	0000	0044	4952	2020	2020	2053	5953DIR.....SYS
311310	A71D	9642	0A00	1101	FFFF	0000	0000	0000	...B.....
311320	5F00	0000	0053	5953	3720	2020	2053	5953SYS7.....SYS
311330	EB29	210E	0500	1620	FFFF	FFFF	FFFF	FFFF	.)!.....
311340	1E00	0000	0042	4143	4B55	5020	2043	4D44BACKUP..CMD
311350	2A5F	9642	0F00	0322	FFFF	FFFF	FFFF	FFFF	*..B...".....
311360	1000	00DF	0043	4F4D	4D41	4E44	2044	4F43COMMAND.DOC
311370	9642	9642	3700	0A03	0D24	1A01	FFFF	FFFF	.B.B7.....\$.....
311380	0000	0000	0000	0000	0000	0000	0000	0000
311390	0000	0000	0000	0000	0000	0000	0000	0000
3113A0	1000	0088	004E	4557	5553	4552	204B	534DNEWUSER.KSM
3113B0	9642	9642	0500	0920	FFFF	FFFF	FFFF	FFFF	.B.B.....
3113C0	0000	0000	0000	0000	0000	0000	0000	0000
3113D0	0000	0000	0000	0000	0000	0000	0000	0000
3113E0	0000	0000	0000	0000	0000	0000	0000	0000
3113F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 3

DIR/SYS	=	TRACK 11, SECTOR 0
SYS7/SYS	=	TRACK 16, SECTOR 5
BACKUP/CMD	=	TRACK 03, SECTOR 5
COMMAND/DOC	=	TRACK 0A, SECTOR 0 (EXTENT 1)
		TRACK 0D, SECTOR 5 (EXTENT 2)
		TRACK 1A, SECTOR 0 (EXTENT 3)
NEWUSER/KSM	=	TRACK 09, SECTOR 5

(FPDE/FXDE SECTOR 3)

(figure A3.5)

311400	5F00	0000	0053	5953	3020	2020	2053	5953SYS0.....SYS
311410	EB29	210E	0F00	0022	FFFF	FFFF	FFFF	FFFF	.)!....."
311420	5F00	0000	0053	5953	3820	2020	2053	5953SYS8.....SYS
311430	EB29	210E	0500	1700	FFFF	FFFF	FFFF	FFFF	.)!.....
311440	1E00	0000	0050	4154	4348	2020	2043	4D44PATCH...CMD
311450	2A5F	9642	0500	0500	FFFF	FFFF	FFFF	FFFF	*..B.....
311460	0000	0000	0000	0000	0000	0000	0000	0000
311470	0000	0000	0000	0000	0000	0000	0000	0000
311480	0000	0000	0000	0000	0000	0000	0000	0000
311490	0000	0000	0000	0000	0000	0000	0000	0000
3114A0	0000	0000	0000	0000	0000	0000	0000	0000
3114B0	0000	0000	0000	0000	0000	0000	0000	0000
3114C0	0000	0000	0000	0000	0000	0000	0000	0000
3114D0	0000	0000	0000	0000	0000	0000	0000	0000
3114E0	0000	0000	0000	0000	0000	0000	0000	0000
3114F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 4

SYS0/SYS = TRACK 00, SECTOR 5
 SYS8/SYS = TRACK 17, SECTOR 0
 PATCH/CMD = TRACK 05, SECTOR 0

(FPDE/FXDE SECTOR 4)

(figure A3.6)

311500	5F00	0000	0053	5953	3120	2020	2053	5953SYS1.....SYS
311510	EB29	210E	0500	1000	FFFF	FFFF	FFFF	FFFF	.)!.....
311520	5F00	0000	0053	5953	3920	2020	2053	5953SYS9.....SYS
311530	EB29	210E	0500	1720	FFFF	FFFF	FFFF	FFFF	.)!.....
311540	1400	0000	0044	4F4C	4320	2020	2044	5652DOLC....DVR
311550	2A5F	9642	0500	0520	FFFF	FFFF	FFFF	FFFF	*..B.....
311560	1000	0022	0056	544F	5320	2020	2045	5054	...".VTOS....EPT
311570	9642	9642	0E00	1B01	1F20	FFFF	FFFF	FFFF	.B.B.....
311580	1000	00FC	0046	4541	5455	5245	5344	4F43FEATURESDOC
311590	9642	9642	0B00	0C02	FFFF	FFFF	FFFF	FFFF	.B.B.....
3115A0	0000	0000	0000	0000	0000	0000	0000	0000
3115B0	0000	0000	0000	0000	0000	0000	0000	0000
3115C0	0000	0000	0000	0000	0000	0000	0000	0000
3115D0	0000	0000	0000	0000	0000	0000	0000	0000
3115E0	0000	0000	0000	0000	0000	0000	0000	0000
3115F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 5

SYS1/SYS = TRACK 10, SECTOR 0
 SYS9/SYS = TRACK 17, SECTOR 5
 DOLC/DVR = TRACK 05, SECTOR 5
 VTOS/EPT = TRACK 1B, SECTOR 0 (EXTENT 1)
 TRACK 1F, SECTOR 5 (EXTENT 5)
 FEATURES/DOC = TRACK 0C, SECTOR 0

APPENDIX A VTOS 3.0 DIRECTORY

(FPDE/FXDE SECTOR 5) (figure A3.7)

311600	5F00	0000	0053	5953	3220	2020	2053	5953SYS2.....SYS
311610	EB29	210E	0500	1020	FFFF	FFFF	FFFF	FFFF	.)!.....
311620	0000	0000	0000	0000	0000	0000	0000	0000
311630	0000	0000	0000	0000	0000	0000	0000	0000
311640	1400	0000	0050	5220	2020	2020	2044	5652PR.....DVR
311650	2A5F	9642	0500	0600	FFFF	FFFF	FFFF	FFFF	*..B.....
311660	1000	00BD	0056	544F	5320	2020	204B	534DVTOS.....KSM
311670	9642	9642	0100	1900	FFFF	FFFF	FFFF	FFFF	.B.B.....
311680	0000	0000	0000	0000	0000	0000	0000	0000
311690	0000	0000	0000	0000	0000	0000	0000	0000
3116A0	0000	0000	0000	0000	0000	0000	0000	0000
3116B0	0000	0000	0000	0000	0000	0000	0000	0000
3116C0	0000	0000	0000	0000	0000	0000	0000	0000
3116D0	0000	0000	0000	0000	0000	0000	0000	0000
3116E0	0000	0000	0000	0000	0000	0000	0000	0000
3116F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 6

SYS2/SYS = TRACK 10, SECTOR 5
 PR/DVR = TRACK 06, SECTOR 0
 VTOS/KSM = TRACK 19, SECTOR 0

(FPDE/FXDE SECTOR 6) (figure A3.8)

311700	5F00	0000	0053	5953	3320	2020	2053	5953SYS3.....SYS
311710	EB29	210E	0500	1200	FFFF	FFFF	FFFF	FFFF	.)!.....
311720	0000	0000	0000	0000	0000	0000	0000	0000
311730	0000	0000	0000	0000	0000	0000	0000	0000
311740	1400	0000	0052	5332	3332	2020	2044	5652RS232...DVR
311750	2A5F	9642	0500	0620	FFFF	FFFF	FFFF	FFFF	*..B.....
311760	1000	00E4	0042	4153	4943	2020	2044	4F43BASIC...DOC
311770	9642	9642	0C00	2002	FFFF	FFFF	FFFF	FFFF	.B.B.....
311780	0000	0000	0000	0000	0000	0000	0000	0000
311790	0000	0000	0000	0000	0000	0000	0000	0000
3117A0	0000	0000	0000	0000	0000	0000	0000	0000
3117B0	0000	0000	0000	0000	0000	0000	0000	0000
3117C0	0000	0000	0000	0000	0000	0000	0000	0000
3117D0	0000	0000	0000	0000	0000	0000	0000	0000
3117E0	0000	0000	0000	0000	0000	0000	0000	0000
3117F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 7

SYS3/SYS = TRACK 12, SECTOR 0
 RS232/DVR = TRACK 06, SECTOR 5
 BASIC/DOC = TRACK 20, SECTOR 0

APPENDIX A VTOS 3.0 DIRECTORY

(FPDE/FXDE SECTOR 7) (figure A3.9)

311800	5F00	0000	0053	5953	3420	2020	2053	5953SYS4.....SYS
311810	EB29	210E	0500	1220	FFFF	FFFF	FFFF	FFFF	.)!.....
311820	0000	0000	0000	0000	0000	0000	0000	0000
311830	0000	0000	0000	0000	0000	0000	0000	0000
311840	1400	0000	004B	534D	2020	2020	2044	5652KSM.....DVR
311850	2A5F	9642	0500	0700	FFFF	FFFF	FFFF	FFFF	*..B.....
311860	1000	007F	0050	454E	4349	4C20	2046	4958PENCIL..FIX
311870	9642	9642	0600	1801	FFFF	FFFF	FFFF	FFFF	.B.B.....
311880	0000	0000	0000	0000	0000	0000	0000	0000
311890	0000	0000	0000	0000	0000	0000	0000	0000
3118A0	0000	0000	0000	0000	0000	0000	0000	0000
3118B0	0000	0000	0000	0000	0000	0000	0000	0000
3118C0	0000	0000	0000	0000	0000	0000	0000	0000
3118D0	0000	0000	0000	0000	0000	0000	0000	0000
3118E0	1000	004A	0049	4E49	5420	2020	204A	434C	...J.INIT....JCL
3118F06	9642	9642	0200	1920	FFFF	FFFF	FFFF	FFFF	.B.B.....

Relative sector 8

SYS4/SYS	=	TRACK 12, SECTOR 5
KSM/DVR	=	TRACK 07, SECTOR 0
PENCIL/FIX	=	TRACK 18, SECTOR 0
INIT/JCL	=	TRACK 19, SECTOR 5

(FPDE/FXDE SECTOR 8) (figure A3.10)

311900	5F00	0000	0053	5953	3520	2020	2053	5953SYS5.....SYS
311910	EB29	210E	0500	1300	FFFF	FFFF	FFFF	FFFF	.)!.....
311920	0000	0000	0000	0000	0000	0000	0000	0000
311930	0000	0000	0000	0000	0000	0000	0000	0000
311940	1E00	0000	004B	5352	2020	2020	2043	4D44KSR.....CMD
311950	2A5F	9642	0500	0720	FFFF	FFFF	FFFF	FFFF	*..B.....
311960	1000	0014	0047	454E	4552	414C	2044	4F43GENERAL.DOC
311970	9642	9642	0800	2121	FFFF	FFFF	FFFF	FFFF	.B.B...!!.....
311980	0000	0000	0000	0000	0000	0000	0000	0000
311990	0000	0000	0000	0000	0000	0000	0000	0000
3119A0	1000	0051	0055	5449	4C49	5459	2044	4F43	...Q.UTILITY.DOC
3119B0	9642	9642	2200	1C06	FFFF	FFFF	FFFF	FFFF	.B.B".....
3119C0	0000	0000	0000	0000	0000	0000	0000	0000
3119D0	0000	0000	0000	0000	0000	0000	0000	0000
3119E0	0000	0000	0000	0000	0000	0000	0000	0000
3119F06	0000	0000	0000	0000	0000	0000	0000	0000

Relative sector 9

SYS5/SYS	=	TRACK 13, SECTOR 0
KSR/CMD	=	TRACK 07, SECTOR 5
GENERAL/DOC	=	TRACK 21, SECTOR 5
UTILITY/DOC	=	TRACK 1C, SECTOR 0

APPENDIX B

APPENDIX B

This originally appeared in the OCTUG newsletter. OCTUG is the "Orange County TRS-80 Users' Group". It is one of the finer TRS-80 clubs and its newsletter is an outstanding publication. For membership information write:

OCTUG
2531 E. COMMONWEALTH AVE.
FULLERTON, CA 92631

SERVICING THE TRS-80 DISK DRIVE (Shugart SA400) By Don Necker

Unless you're prepared to work on the unit in a relatively clean area, free from dirt and lint, it would be best to leave the unit alone. The tools you will need for this "light maintenance" are a Phillips and standard screwdriver, a small wrench for number four and six hex nuts, a can of Freon spray cleaner (must state on can: SAFE FOR ALL PLASTICS), a small amount of isopropyl (rubbing) alcohol, a couple of lint free wipers and a little silicone light lubricant (such as Garcia Reel-lube). If there is an apparent power supply overheating problem you will need a small (approximately 40 watt) soldering iron and solder and a dab of heat sink grease.

Remove the outside cover by removing the four Phillips screws. Then, by removing the three screws which attach the Shugart drive assembly to the rear and bottom frames, the drive assembly may be slid forward out of engagement with the 34 pin connector. This facilitates disconnecting the 4 pin power plug. The drive assembly is now free from the frame and power supply.

In handling the drive avoid contaminating the belts and pulleys with body oils from your fingers. By removing the two small screws in the drive's large circuit board and disconnecting the two connectors, the board may be removed, exposing the disk drive mechanism.

Examine the drive mechanism for evidence of dirt, lint or other foreign material. The read/write head assembly may be slid fore and aft out of the drive cam detent with slight finger pressure to check for binding. Freon spray should be used to wash out any foreign matter throughout the mechanism. The read/write head, felt pressure pad and the LED and sensor faces should be wiped clean with alcohol on a wiper. Be careful not to apply pressure to these items which would knock them out of alignment.

The use of metal objects is not recommended since they may scratch the critical surfaces. A slight film of lubricant should be applied to the two round head-slider guides and at a couple of spots along the cam drive grooves. The two drive belts and pulleys should be wiped clean using freon and wipers. A dab of lubricant in each of the front door latch grooves completes the servicing of the drive.

If you have been having operating problems which occur after the unit has been on for a while, it may be a heat dissipation problem in the power supply. If so, check the two three-terminal regulator ICs

APPENDIX B

which are fastened to the inside of the rear frame. The IC metal surfaces should be tight against the frame. A mica insulator with heat sink grease ON BOTH SIDES, should be visible between the ICs and the frame.

Some units have been found with number four nylon screws which have been stripped. These number four screws should be replaced with number six button head nylon screws (Number six metal nuts may be used if installed on the inside surface).

After the old screws are removed, re-install the mica insulator with added heat sink grease on both sides. Reform the IC leads to position the IC against the frame. Install the new screws making sure you don't strip the threads. Resolder the IC leads where they enter the board. It may be necessary to remove the board from the frame if your soldering iron is too large or your hand is not steady enough.

The disassembly procedure is reversed to reassemble the unit. All connectors have locking features to assure proper alignment and orientation.



APPENDIX B

SUGGESTED READING

I don't know of another book on data recovery, if one existed, I'd certainly recommend it. However, there are a great number of excellent publications currently on the market about computers in general. If you would like to become better at what you do and don't want to spend time re-inventing the wheel, try learning from these authors. I have found their books instructional, easy to read, and a cut above average. So why watch another re-run of the 'Flintstones' when you can read a good book?

'HOW TO PROGRAM MICROCOMPUTERS'

Author: William Barden, Jr.

Publisher: Howard W. Sams & CO.

TRS-80 ASSEMBLY LANGUAGE PROGRAMMING

Author: William Barden, Jr.

Publisher: Radio Shack

COMPUTER ARCHITECTURE

Author: Caxton C. Foster

Publisher: Van Nostrand Reinhold Company

INTRODUCTION TO COMPUTER PROGRAMMING

with the BASIC Language

Author: Harvey M. Deitel

Publisher: Prentice-Hall, Inc.

THE BASIC HANDBOOK

An Encyclopedia of the BASIC Computer Language

Author: David A. Lien

Publisher: Compusoft Publishing

LEARNING LEVEL II

Learning TRS-80 Level II BASIC

COMPUSOFT LEARNING SERIES

Author: David A. Lien

Publisher: Compusoft Publishing

MAKING SYSTEMS WORK

The psychology of Business Systems

Author: William C. Ramsgard

Publisher: John Wiley & Sons

APPENDIX B

MURPHY AND HIS DAMNED LAW: Whatever can go wrong, will.

The trouble with a cliché is that it's true. In the interest of increasing your knowledge of computers I feel it my duty to expose you to the Truths of the "Way Things Really Are". Since the legendary Murphy is no longer with us, a victim of his own laws, (Mr. Murphy owned and operated a hand grenade repair business). I have had to rely on Fenwyler T. Murphy, his nephew and executor of the Murphy estate as a source for the following material. (Fenwyler Murphy is also the director of the Murphy Memorial Foundation For The Study of Known Phenomena.)

In the interest of preserving space I have listed only those laws which apply most directly to computers and programming.

COROLLARIES:

GUTTERSON'S LAWS:

Any programming project that begins well, ends badly.
Any programming project that begins badly, ends worse.

KLIENERUNNER'S COROLLARIES:

If a programming task looks easy, it's tough.
If a programming task looks tough, it's damn well impossible.

MUNGBRIGHT'S LAWS:

Any given program, when running, is obsolete.
Any given program costs more and takes longer.
If a program is useful, it will have to be changed.
If a program is useless, it will have to be documented.
Any given program will expand to fill all available memory.
The value of a program is inversely proportional to the weight of its output.
Program complexity grows until it exceeds the capability of the programmer who must maintain it.
Not until a program is in release for six months will the most harmful error be discovered.
Machine independent code, isn't.
Adding manpower to a late software project makes it later.
The effort required to correct software problems increases geometrically with time.

FARVOUR'S LAW:

There is always one more bug.

BRUNK'S LAW:

If a listing has a beginning it has an end.

ZEPPEMIER'S COROLLARY:

The last 4 pages of a critical listing will be lost.

PENNINGTON'S OBSERVATION:

The probability that a given program will perform to expectations is inversely proportional to the programmers' confidence in his ability to do the job.

APPENDIX B

ORDERING INFORMATION

If your favorite software dealer does not stock NEWDOS+ with SUPERZAP, the following APPARAT NEW DOS Distributors will be more than happy (grateful, in fact) to fill his order, instantly.

Apparat Inc.
7310 East Princeton
Denver, Colorado 80237
(303) 758-7275

IJG Computer Services
569 N. Mountain Ave - Suite B
Upland, California 91786 U.S.A.
(714) 982-7829

Miller Microcomputer Services
61 Lake Shore Road
Natick, Massachusetts 01760
(617) 653-6136

If you would like additional copies of this book they may be purchased through your local book seller, software dealer, or direct from IJG, Apparat or Miller, listed above.

NOTICE

This book is the beginning of a series of publications specifically for the TRS-80, currently in progress. The following titles are planned for publication in late 1980.

VOLUME II TRS-80 INFORMATION SERIES
BASIC LISTED AND COMMENTED.

VOLUME III TRS-80 INFORMATION SERIES
'DOS' LISTED AND COMMENTED

VOLUME IV TRS-80 INFORMATION SERIES
GUIDE TO HARDWARE EXPANSION AND MODIFICATION

APPENDIX B

"SEARCH 1.0"

=====

'SEARCH' is a BASIC language program that will search a disk file for any byte combination up to 255 bytes. The user is prompted to enter the file specification and line printer option.

It will return the relative sector and the starting byte (in decimal), in which a match was found to the display and/or the line printer.

The input requires a 2 character hexadecimal input for each search value. After each 2 character input is 'ENTER'ed, the input is echoed to the display. Each input is checked for validity. If the input is incorrect, an error message will be flashed on the screen and the user will be prompted to re-enter a valid hexadecimal number.

The disk I/O is 'RANDOM' mode. To conclude the input routine enter 'END' and the search mode is initiated. If the line printer was not specified, the routine will pause after the display is filled and will prompt the user to hold the 'ENTER' key. The screen will be cleared of previously listed matches, and will continue until the routine completes its task or another screen is filled.

Upon completion of the program, "...ALL DONE" is displayed on the video monitor. 'ENTER' must be pressed to continue for another 'RUN'.

CAUTION: Due to the limitation of various disk operating systems, ONLY THE FIRST 255 BYTES OF EACH SECTOR ARE SEARCHED. Byte 256 of any sector is not searched! 'SEARCH' does not span sectors in the search mode. Each sector is searched individually. If the search value(s) are located between loader codes and load addresses the search will not recognize the value, being searched for with the embedded codes.

LIST OF VARIABLE NAMES USED AND FUNCTIONS

A\$ - Sector buffer
B\$ - Sector buffer comparison string - makes system
comptable with SUPERDOS 1.0
C\$ - Instring position counter
CK\$ - Comparison string for 'INSTRING' routine.
CV\$ - Hexadecimal characters
DR\$ - Drive specification
FS\$ - File specification
I\$ - Hexadecimal input value
II\$ - 'Echo' string
IK\$ - Inkey string
L - Instring position of 'search value'
LP - line print switch
LP\$ - line printer input to set switch
N - Hex conversion routine variable
N1 - Hex conversion routine variable
PZ - Display print position
SC\$ - Contains search value(s)
T(1) - Hex conversion routine variable
T(2) - Hex conversion routine variable
X - Record number in 'GET'
X1 - Loop counter
X2 - Loop counter

APPENDIX B

```

100  REM *****
250  REM **          SEARCH          08/30/79          **
200  REM **          BY H.C. PENNINGTON          **
250  REM **-----**
300  REM **  'SEARCH' WILL FIND ANY HEX STRING IN **
350  REM **  A DISK FILE.  INPUTS ARE 2 CHARACT- **
400  REM **  ER HEX NUMBERS.  'END' TERMINATES **
450  REM **  THE INPUT MODE AND INITIATES THE **
500  REM **  SEARCH. **
550  REM *****
600  REM
650  REM *****
700  REM  INITIALIZE AND FILE SPECIFICATION INPUT
750  REM *****
800  CLS:
    CLEAR 1000:
    CV$="0123456789ABCDEF"
850  PRINT@192,:
    INPUT" ENTER FILE SPEC: ";FS$
900  DR$="0":
    INPUT" ENTER DRIVE (0 - 3): "; DR$
950  IF VAL(DR$)<0 OR VAL(DR$)>3 THEN GOTO 800
1000 INPUT"DO YOU WISH OUTPUT TO LINE PRINTER (Y - N)";LP$
1050 IF LEFT$(LP$,1)="Y" THEN LP=1
1100 FS$=FS$+" "+DR$
1150 CLS:PRINT@192,
    "INPUT ALL VALUES AS 2 CHARACTER HEXADECIMAL NUMBERS."
1200 PRINT
    "ENTER EACH 2 CHARACTER INPUT.  WHEN FINISHED ENTER 'END'"
1250 PRINT"EXAMPLE '01' = 1          '0A' = A"
1300 PRINT@ 384, STRING$(63,140)
1350 REM
1400 REM *****
1450 REM          INPUT' HEXADECIMAL VALUES
1500 REM          AND TEST INPUT FOR CORRECT ENTRY
1550 REM *****
1600 PRINT@512,,:
    INPUT I$:
    IF I$="END" THEN GOTO 2350:' 512
1650 IF LEN(I$) > 2 OR LEN(I$) < 2 THEN GOTO 3250
1700 T(1)=INSTR(CV$,LEFT$(I$,1)):
    IF T(1)=0 THEN GOTO 3250
1750 T(2)=INSTR(CV$,RIGHT$(I$,1)):
    IF T(2)=0 THEN GOTO 3250
1800 II$=II$+" "+I$:
    PRINT@ 640,II$:
    PRINT@ 512,STRING$(63,32):
    PRINT@ 448,;

```

APPENDIX B

```

1850 REM
1900 REM *****
1950 REM      CONVERT INPUT TO CHARACTER SEARCH STRING
2000 REM *****
2050 X = 1:
      CK$ = LEFT$(I$,1)
2100 N = INSTR( CV$,CK$):
      N=N-1
2150 IF X=1 THEN N1 = N * 16:
      X=X+1:
      CK$ = RIGHT$(I$,1):
      GOTO 2100
2200 N = N+N1
2250 SC$ = SC$ + CHR$(N):
      GOTO 1600

2300 REM
2350 REM *****
2360 REM      SEARCH ROUTINE
2370 REM *****
2400 X=1:
      CLS:
      PRINT@ 0,"SEARCHING RELATIVE SECTOR: 0 ";FS$;:
      PZ=128
2450 IF LP=1 THEN LPRINT
      "SEARCHING FILE: ";FS$:LPRINT"SEARCH VALUE: ";II$:
      LPRINT STRING$(50,"="):
      LPRINT" "
2500 OPEN"R",1,FS$
2550 FIELD1, 255 AS A$:
      B$=""
2600 GET 1,X:
      C=1:
      PRINT@27,X;
2650 L=INSTR(C,A$,SC$):
      IF PZ>=960 AND LP=0 THEN
        GOSUB 3900:
        PRINT@ 64,STRING$(40,32);:
        PZ=128
      ELSE IF PZ >=960 THEN PZ=128
2700 IF L>0 THEN C=C+L+1:
      PRINT@ PZ,
      " MATCH = RELATIVE SECTOR"; X-1;TAB(30)"BYTE ="; L;" ";:
      PZ=PZ + 64
2750 IF L>0 AND LP=1 THEN LPRINT
      "MATCH = RELATIVE SECTOR ";X-1;TAB(31)"BYTE ="; L
2800 IF A$=B$ OR A$=STRING$(255,0) THEN 3000
2850 IF L>0 AND C<255 GOTO 2650
2900 B$=A$
2950 C=1:
      X=X+1:
      GOTO 2600
3000 PRINT@67, ".... ALL DONE ";:
3010 IK$=INKEY$:
      IF IK$="" THEN 3010
3020 RUN

```


APPENDIX B

```

3050 REM
3100 REM *****
3150 REM          INPUT ERROR ROUTINE
3200 REM *****
3250 FOR X1= 1 TO 9:
    PRINT@0,:
    PRINT:PRINT:PRINT:PRINT
3300 PRINT@ 270, "YOU HAVE ENTERED AN INCORRECT VALUE."
3350 PRINT TAB(24)"PLEASE DO AGAIN.";
3400 FOR X2= 1 TO 80:NEXT
3450 PRINT@ 270,STRING$(36,32):
    PRINT;
3500 FOR X2= 1 TO 50:NEXT
3550 NEXT X1:
    PRINT @512,STRING$(63,32):
    PRINT@ 512,;
3600 GOTO 1600
3700 REM
3750 REM *****
3800 REM          PAUSE & FLASH MESSAGE ROUTINE
3850 REM *****
3900 IK$=INKEY$:
3950 PRINT@ 64,"HOLD ENTER TO CONTINUE";
4000 FOR X1 = 1 TO 50
4050 IK$=INKEY$:
    IF IK$="" THEN NEXT X1
4100 IF IK$=CHR$(13) THEN 4400
4150 PRINT@ 64, STRING$(25,32)
4200 FOR X1 = 1 TO 50
4250 IK$=INKEY$:
    IF IK$="" THEN NEXT X1
4300 IF IK$=CHR$(13) THEN 4400
4350 GOTO 3950
4400 FOR X1 = 1 TO 13:
    PRINT:
    NEXT X1:
    RETURN

```

APPENDIX B

```

650      *****
700      INITIALIZE AND FILE SPECIFICATION INPUT
750      *****
800      Clear the screen
      Clear string space
      Initialize CV$ with alphanumeric characters
850      Set print position
      Input file specification
900      Set drive specification default value
      Input drive specification
950      Test input
1000     Input output mode (line printer or display only)
1050     Test input
1100     Concatenate file specification
1150     Clear the screen
      Print instruction message to screen
1200     Continue message
      Continue message
1250     Continue message
1300     Print graphics line to screen
1350
1400     *****
1450     INPUT HEXADECIMAL VALUES
1500     AND TEST INPUT FOR CORRECT ENTRY
1550     *****
1600     Set screen print position
      Input hexadecimal value
      Test for end of input
1650     Test input for valid length
1700     Get decimal value of left side of I$
      Test for valid input
1750     Get decimal value of right side of I$
      Test for valid input
1800     Concatenate 'echo' string
      Set screen print position of 'echo' string & print
      Clear previous input from display
      Set print position for next user input

```

APPENDIX B

```

1850
1900      *****
1950      CONVERT INPUT TO CHARACTER SEARCH STRING
2000      *****
2050  Set instring counter
      Set CK$ to first search character
2100  Search hex character string for position of CK$
      Get correct hexadecimal multiplier
2150  If first pass then get left side hex value
      Increment instring counter
      Set CK$ to second search character
      Do it again
2200  Add decimal values
2250  Concatenate search string
      Get next user input
2300
2350      *****
2360      SEARCH ROUTINE
2370      *****
2400  Set record number to 1
      Clear the screen
      Print message to display
      Set display print position
2450  Check line print switch
      Print header message on line printer
      Continue message
      Continue message
2500  Open file
2550  Field sector buffer
      Set comparison string to null
2600  Get sector
      Set 'start instring search' position counter
      Print current sector being searched to screen
2650  Search sector for match
      Check print position and re-set if necessary
      If screen full & line printer switch not set
          then go to 'pause' routine
      Set screen print position
      Re-set screen print position if screen full
2700  If match found then increment instring position counter
      Set screen print position
      Print message
      Increment screen print position
2750  If line printer switch set
          then line print message
2800  Check for end of file - if end conclude program
2850  Is instring search finished? - do again if not finished
2900  Set 'EOF' comparison string
2950  Re-set instring search counter to 1
      Increment record number
      Do again
3000  Print "ALL DONE" message
3010  Inkey routine to lock-out reinitialization of program
      Check IK$ for input
3020  'RUN' program again

```

APPENDIX B

```

3050
3100 *****
3150 INPUT ERROR ROUTINE
3200 *****
3250 Set loop for 'flashing' error message
      Set print position for message
      Clear previous message
3300 Print message
3350 Continue message
3400 Delay loop
3450 Clear message from screen

3500 Set delay loop for 'flash' off
3550 Loop
      Clear message from display
      Set print position for user input
3600 Return to input routine
3700
3750 *****
3800 PAUSE & FLASH MESSAGE ROUTINE
3850 *****
3900 Set Inkey string
3950 Print message
4000 Set delay loop
4050 Set Inkey string
      Test for input
4100 If input is carriage return then go to clear screen
4150 Clear message from screen
4200 Delay loop
4250 Set Inkey string
      Test Inkey string
4300 If input is carriage return then go to clear screen
4350 Flash message again
4400 Set loop to clear screen
      print nulls to screen
      do again
      return to calling routine

```